THE UNIVERSITY
of EDINBURGH

Ultra Low Power Cooperative Branch Prediction

Matthew Bielby

Doctor of Philosophy

Institute of Computing Systems Architecture

School of Informatics

University of Edinburgh

2015

Abstract

Branch Prediction is a key task in the operation of a high performance processor. An inaccurate branch predictor results in increased program run-time and a rise in energy consumption. The drive towards processors with limited die-space and tighter energy requirements will continue to intensify over the coming years, as will the shift towards increasingly multicore processors. Both trends make it increasingly important and increasingly difficult to find effective and efficient branch predictor designs.

This thesis presents savings in energy and die-space through the use of more efficient cooperative branch predictors achieved through novel branch prediction designs. The first contribution is a new take on the problem of a hybrid dynamic-static branch predictor allocating branches to be predicted by one of its sub-predictors. A new bias parameter is introduced as a mechanism for trading off a small amount of performance for savings in die-space and energy. This is achieved by predicting more branches with the static predictor, ensuring that only the branches that will most benefit from the dynamic predictor's resources are predicted dynamically. This reduces pressure on the dynamic predictor's resources allowing for a smaller predictor to achieve very high accuracy. An improvement in run-time of 7-8% over the baseline BTFN predictor is observed at a cost of a branch predictor bits budget of much less than 1KB.

Next, a novel approach to branch prediction for multicore data-parallel applications is presented. The Peloton branch prediction scheme uses a pack of cyclists as an illustration of how a group of processors running similar tasks can share branch predictions to improve accuracy and reduce runtime. The results show that sharing updates for conditional branches across the existing interconnect for I-cache and D-cache updates results in a reduction of mispredictions of up to 25% and a reduction in run-time of up to 6%. McPAT is used to present an energy model that suggests the savings are achieved at little to no increase in energy required. The technique is then extended to architectures where the size of the branch predictors may differ between cores. The results show that such heterogeneity can dramatically reduce the die-space required for an accurate branch predictor while having little impact on performance and up to 9% energy savings. The approach can be combined with the Peloton branch prediction scheme for reduction in branch mispredictions of up to 5%.

Acknowledgements

First of all I would like to thank my supervisors, Björn Franke and Nigel Topham, for all their help over the course of my thesis.

I would also like to thank my co-workers at the University of Edinburgh, especially those in my office and those who were kind enough to read my draft paper submissions. My thanks also go to my other friends who supported me over the last few years, especially Chris and all the members of Edinburgh Genbukan Iaido Dojo.

I would like to give special thanks to all the flat mates I have had during my time in Edinburgh, especially Chris and Rachel.

Finally, I would like to thank my family for supporting me through my time at Edinburgh University.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified. Some of the material used in this thesis has been published in the following paper:

- **Matthew Bielby**, Miles Gould, and Nigel Topham. 2012. Design Space Exploration of Hybrid Ultra Low Power Branch Predictors. *In Proceedings of the 25th international conference on Architecture of Computing Systems* (ARCS'12).

(*Matthew Bielby*)

To Olga, Steve, Chris and Rachel.

Table of Contents

Abbreviations

BHT Branch History Table

BIU Branch Identification Unit

BNFT Backwads Not-taken Forwards Taken

BPU Branch Prediction Unit

BTAC Branch Target Address Cache

BTB Branch Target Buffer

BTFN Backwards Taken Forwards Not-taken

CBP Complementary Branch Predictor

CHS Compiler-guided History Stack

CMP Chip Multiprocessor

CPU Central Processing Unit

ED Energy Delay product

ED^2 Energy Delay Delay product

GHR Global History Register

GPU Graphics Processing Unit

IPC Instructions Per Cycle

ISA Instruction Set Architecture

MMU Memory Management Unit

NoC Network on Chip

NOP No Operation

PHT Pattern History Table

PPD Prediction Probe Detector

RAS Return Address Stack

SBHSR Shared Branch History Shift Register

SMT Simultaneous Multithreading

SoC System on Chip

SPHT Shared Pattern History Table

VLIW Very Long Instruction Word

XOR Exclusive OR

ZOL Zero Overhead Loop

Chapter 1

Introduction

The development of modern processors has seen a consistent rise in the power and speed available, enshrined in Moore's Law. With multicore and embedded systems becoming increasingly more common in an increasingly diverse number of settings the field of processor microarchitecture design is focused on maintaining and improving performance in the face of novel power, energy and cost challenges provided by these platforms.

Processors make use of a pipeline structure to allow for a higher throughput of instructions, allowing for high speeds that are now required. Branch predictors are an important component in assuring the speed and energy efficiency of the modern processor. Branch predictors are used to keep the processor pipeline filled with the correct instructions, with high accuracy being important not only for speed of program execution but also the energy required. This thesis looks at how proven branch prediction technologies can be combined and enhanced to provide new prediction mechanisms suitable for embedded and multicore applications.

This thesis takes a hardware based approach to the problem of branch prediction. As such, most of the discussions are around hardware based problems and solutions and while the introduction and background chapters introduce the key concepts of hardware branch prediction, some basic knowledge of computer hardware is assumed. Furthermore, while this thesis does not seek to explicitly identity problems of branch prediction that are presented through the level of abstraction available to a compiler, there are some points at which the assistance of information from the compiler is invaluable when presented to the correct hardware elements. As such there will be a limited discussion of the role that the compiler can play in modern low power branch prediction techniques. The hardware is considered from an architectural and micro-

architectural stand point. As a result this thesis does not get down to the level of a hardware description language and gate level layouts.

This thesis uses a cycle accurate simulator to model the performance of the processor, which is then coupled with energy estimation tools such as CACTI [71]. The architectures examined are generally low power, small area, in-order ARM cores for the embedded environment.

1.1 Pipelining

Pipelining is a technique used in modern processors to increase efficiency of resource use and to reduce the time required to execute a series of instructions. In order to fill the pipeline it is necessary to fetch one instruction per cycle. Usually this is simply a case of fetching the next instruction to execute and issuing it. However, in the case of branches or jumps it is unclear what path the program will take though its code until the branch or jump instruction is most of the way through the pipeline. This introduces bubbles, or stall cycles.

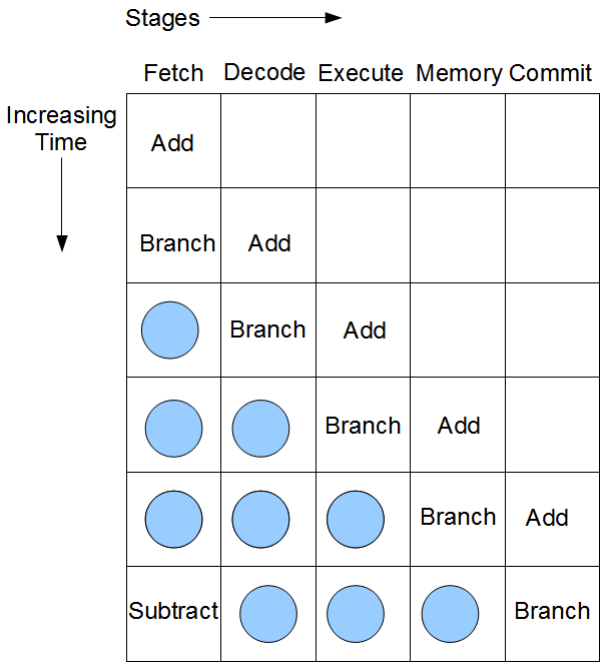

Figure 1.1: Each column shows a different stage of the pipeline. Each row shows the state of the progress of instructions through the pipeline every cycle. The blue circles show stall cycles, where no instruction is executed in the given pipeline stage. Without a branch predictor stall cycles are added in cycles 3-5, increasing execution time.

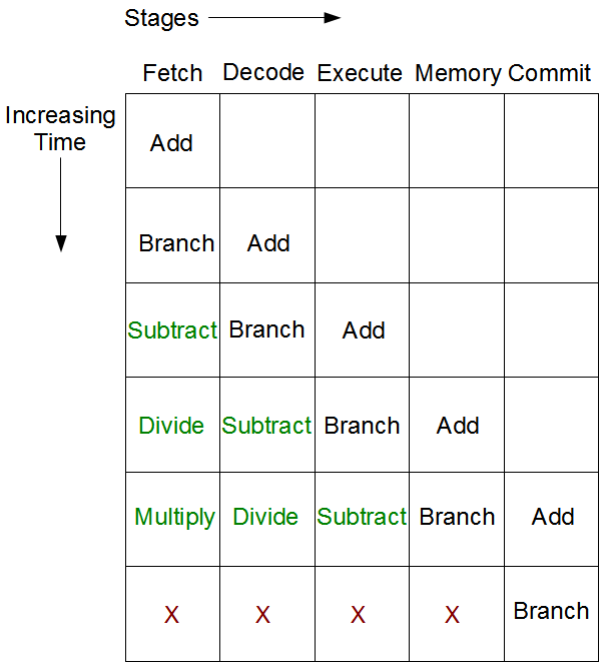

Figure 1.2: With a branch predictor the stall cycles are avoided by following the predicted branch outcome. The green instructions are those fetched on the prediction from the branch predictor. However, if the branch turns out to be mispredicted the pipeline must be flushed and filled anew, slowing the program down more than simply waiting through the stall cycles. In this example it turns out the branch was mispredicted, so the green instructions are removed and it takes another cycle to start the next instruction. This results in a total of 4 cycles delay.

Branch predictors are used to predict whether a branch will be taken or not taken as well as giving a prediction of the next instruction to be fetched. When correctly predicted this removes the stall cycles, increasing the instruction throughput and speeding up program execution. An example of this is presented in figures 1.1 and 1.2.

1.2 Properties Of A Branch

Branches can be classified based on several different orthogonal attributes which describe when and how a branch is taken. These attributes contribute to how difficult it is to predict a branch and are often specially targeted by branch prediction techniques for different prediction methods based on their combination of attributes.

A branch can be classified by when it is taken as either conditional or unconditional. A conditional branch is one that either has a condition directly encoded into

the instruction, such as branch if this register value is equal to zero, or makes use of status flags that have been set by other instructions, such as branch if the immediately previous add instruction set the overflow flag.

Branches can be classified by their branch target as either direct or indirect. A direct branch has a fix branch target which is either an absolute or relative address. An absolute address is one where the target is written into the PC, while a relative address is calculated by adding an offset to the current PC value. In contrast, an indirect branch is one where the branch target is stored in some intermediate location, typically a memory address. This means that the target of the branch can be changed during execution of the program.

For example, a call instruction to given absolute address would be a direct branch, the address is known statically ahead of time and will not change during execution of the program. Call instructions are typically unconditional branches. Put together this means that call instructions are typically easy to predict from the second time they are encountered as they will likely be unconditionally taken and will be taken to the same target address as last time. In contrast, a return instruction will generally be an unconditional indirect branch. The return is an indirect branch because the target is a value taken from the stack and so may be a different target each time it is encountered, making it much harder to predict.

1.3 Static Vs Dynamic Branch Predictors

Branch prediction techniques can be broadly classed as either static or dynamic. Static branch predictors are those which have set rules about how a branch should be predicted and these rules do not change throughout the execution of a program. As such, every time a branch instruction is seen it will always be predicted in the same way. Static branch predictors require differing amounts of information to make their prediction depending on the specific prediction technique being employed. Generally this means that the branch can only be predicted once the instruction has passed through the decode stage of the pipeline. This often results in several cycles of pipeline stalls, but less than would be seen if the branch was not predicted until the execute stage was complete.

For example, the pipeline shown in figure 1.1 has a 3 cycle stall if no branch predictor is present (while the branch instruction proceeds through the fetch, decode and execute stages) and a 2 cycle stall if a static branch predictor is used (while the in-

struction proceeds through the fetch and decode stages). In modern processors each of these stages can be broken up into multiple sub-stages, meaning that the pipeline stall for a static prediction can be significantly greater.

In contrast, dynamic branch prediction units make use of information that becomes available at run time in order to make their predictions. This means that a branch that is seen many times during the execution of a program can be predicted one way at the start of the execution and in a different way later in the execution. Data collected at run time is used to correlate the behaviour of the branch with the behaviour of other branches. This in turn allows patterns in branch behaviours to be identified at run time and exploited to produce increased accuracy. Furthermore, this allows branches to be predicted with reduced information, allowing branches to be predicted at the start of the pipeline and (in the case of accurate predictions) removing all branch stall cycles.

An example of the impact of a dynamic predictor is shown in figure 1.2. Each of the stall cycles added by the requirement to wait for the branch to be resolved, shown in figure 1.1, are replaced with useful instructions from the correctly predicted program execution path. The branch outcome can be predicted in the fetch stage thanks to the use of structure such at the Branch Target Address Cache (BTAC), which are used to store the addresses of known branches and their branch targets. However, these require the branch to be executed at least once in order to collect information on the branch, so the first time a branch instruction is encountered the dynamic branch predictor makes no prediction so either the branch is predicted not taken or prediction is delayed until a static prediction becomes available.

There are two further important effects of the branch predictor that should be highlighted at this point. In figure 1.1 the lack of available branch prediction leads the processor to stall until the branch is resolved. Doing this means that the performance of a program will be lengthened by these stall cycles for each branch instruction, resulting in a small amount of wasted energy as the processor sits idle. In contrast, figure 1.2 shows the case of a dynamic predictor that has mispredicted. In this case the stall cycles are avoided but the pipeline must be flushed when the misprediction is detected. This results in an increased program execution time and reduced performance (due to the extra 2 cycles introduced by the pipeline flush). Furthermore, the processor has been actively executing the mispredicted instructions until the pipeline is flushed, meaning that increased energy is required over simply stalling until the outcome is known. However, if the prediction had been correct then the program would proceed with no stall cycles, increasing performance and reducing energy consumption as a

result.

1.4 Branch Statistics

The number of branches encountered in a program depends on a large number of factors, including the nature of the application, the programming language used, the compiler used and the ISA. [28] puts the frequency of control flow instructions taken from some sample SPECint2000 [17] benchmarks at between 12% and 25%. As such, any improvement to branch predictor accuracy (and so a reduction in branch stall or misprediction cycles) will have a large impact on the run time of an application.

The different predictor types can have very different accuracy rates. The simplest, static predictors, achieve accuracies of around 75% [3]. The more powerful dynamic 2-level branch predictors achieve accuracies of 90-95% [78]. State-of-the-art hybrid and profiled predictors can achieve accuracies of 97% and even up to or exceeding 99% [56], [5]. The different types of predictors are discussed further in chapter 2.

Modern predictors make use of different variants of large cache-based predictor tables, storing history on past branch outcomes and targets, to aid in the accuracy of predicting future branch outcomes. A generalised example of this is given in figure 1.3. Techniques vary in the amount of history they collect, whether the history is for all branches or on an individual branch level and how counters should be accessed and updated.

The most effective predictors make use of several sub-predictors. These often specialise in being highly accurate in predicting different classes of branches and are then combined through the use of a meta-predictor which chooses which sub-predictor to use for a given prediction. A generalised example of this is given in figure 1.4.

1.5 Impact Of Branch Types On Hardware Requirements

Through considering some of the information already highlighted in sections 1.2, 1.3 and 1.4 it is possible to understand the demands that different branch predictor types place on the hardware. This in turn gives an understanding of the role of each of the components that are found in the different types of branch predictors.

It has already been mentioned that it is desirable for a dynamic predictor to be able to predict a branch outcome and target location while the instruction is still in the fetch stage. For this to be possible a dynamic predictor will generally maintain a cache

structure along the lines of a Pattern History Table (PHT). These are accessed each cycle to check whether there is an entry matching the current instruction (that is one with an cache index and tag hit). If such an entry is found it will contain the expected branch target. An example of this is shown in figure 1.3. The amount of data that is stored, both in terms of the size of each entry and the number of entries, will have a large impact on the accuracy and energy consumption of the dynamic branch predictor.

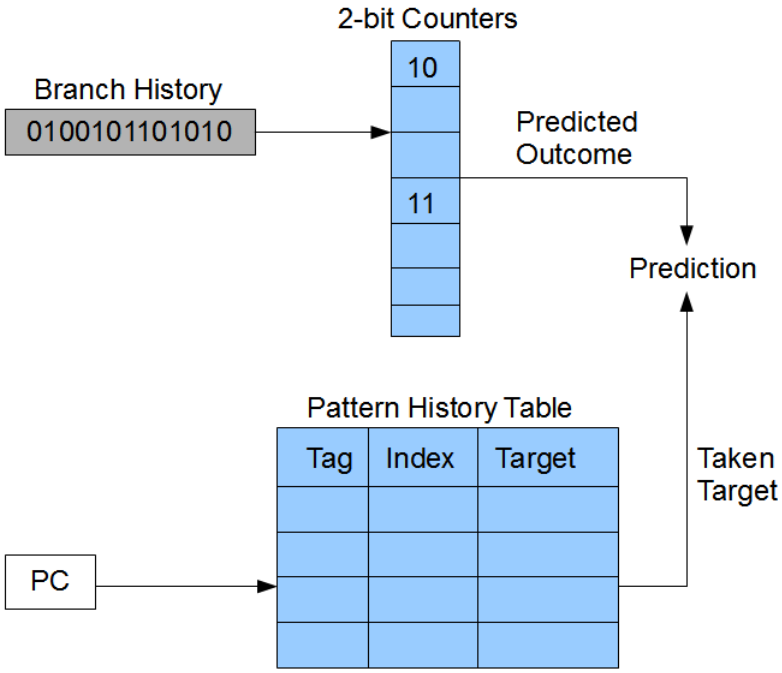

Figure 1.3: A general dynamic two level branch predictor. Outcomes of the past branches are used to pick a 2-bit saturating counter which predicts the branch outcome. The branch PC is used to address the Pattern History Table, which is a large cache structure containing the predicted target for the branch if it is taken, based on the last observed target. Both the outcome and target predictions must be correct to correctly predict the next instruction.

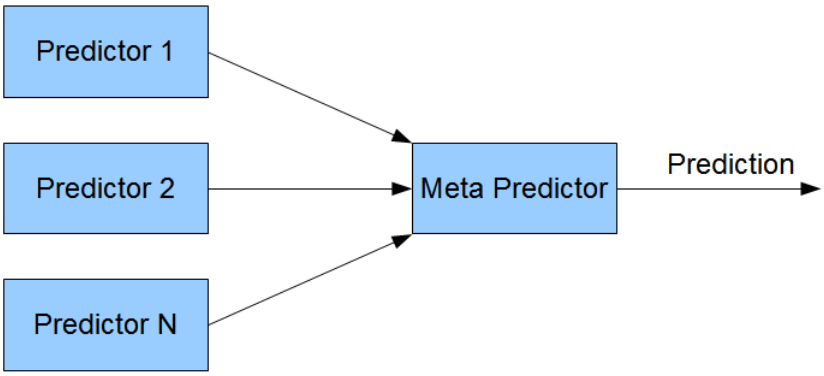

Figure 1.4: A high level concept of a hybrid predictor. Multiple different sub-predictors are used, where each can be a different size, use different indexing methods or even be a completely different kind of predictor. These predictions are then sent to a meta-predictor, which decides which prediction (or combination of predictions) to use as the final branch prediction.

There is pressure against having too many entries in the PHT as this will rapidly increase the die space required and will greatly increase the energy consumption of the predictor. However, having too few entries will result in the cache being unable to store information on enough branches. If the cache does not contain information on a branch when it is queried about the current instruction then the prediction is that the branch is not taken. In effect the predictor assumes that no instructions are branches, or that any branch instructions that do exist are not taken, unless an entry is found in the PHT.

It is possible to reduce the predictor size by reducing the size of each entry in the PHT. This can be done by reducing the tag size or the target size. Reducing the tag size will result in potentially inaccurate matchings where an instruction is incorrectly found to match the branch information that is stored in the PHT, resulting in a branch misprediction. Since the tag is typically formed from the higher instruction bits it is possible to remove some tag bits without impacting accuracy as long as program execution remains in the same area. It is also possible to reduce the entry size by removing bits from the branch target. This has the effect of reducing the amount of PC bits that can be set by the PHT, limiting the range of branch target addresses that can be predicted. Many branches are short range (such as loops) and can be predicted with few bits. However, branches with a longer range (such as calls) would be incorrectly predicted as the update to the PC would not be large enough to reach the required target, resulting in a branch misprediction. Ultimately it is up to the processor designer

to balance off the requirements of predictor size against predictor accuracy.

A similar consideration needs to be made for the structure which makes the prediction as to whether a branch is taken or not taken. In simple branch predictors such as the one just outlined it is possible to predict all branches not-taken unless an entry is found in the PHT. This performs well for unconditional branches or branches with a condition that are generally satisfied such that the branch is taken. However, for conditional branches that have a more even balance of taken or not-taken it is desirable to predict when this will occur. This is often done through accessing a 2 bit counter, the value of which is used to determine the prediction. The manner in which this works is further described in section 2.4. The manner in which the index for accessing these counters and the number of counters available to the predictor both have a large impact on predictor accuracy and predictor energy consumption, just as with the PHT. The possible problems arising from these decisions and their impacts are further discussed in section 1.7.

By varying these properties and others described in 2, it is possible to create different types of predictors which are better at predicting branches with different properties, be they direct/indirect, forward/backward, conditional/unconditional. By putting these predictors together in the right manner it is possible to achieve hybrid predictors, such as in figure 1.4, which are better suited to predicting all types of branches with the best possible accuracy and the lowest possible cost.

A static branch predictor will be able to predict different branches at different times, dependant on the information that is required and when it becomes available. An unconditional branch can be resolved as soon as the branch target becomes available in the decode stage. In contrast, a conditional branch that compares two register values can only be predicted statically once the values to be compared have been read from the register file (in the decode stage) and then compared in the arithmetic logical unit (ALU) in order to evaluate the condition (in the execute stage). A slight variation on this is shown in figure 1.5 where the conditional branch is evaluated on the value of the status flag register which has been set by the previous instruction. As a result the branch outcome can be evaluated as soon as the value has been read out of the register file.

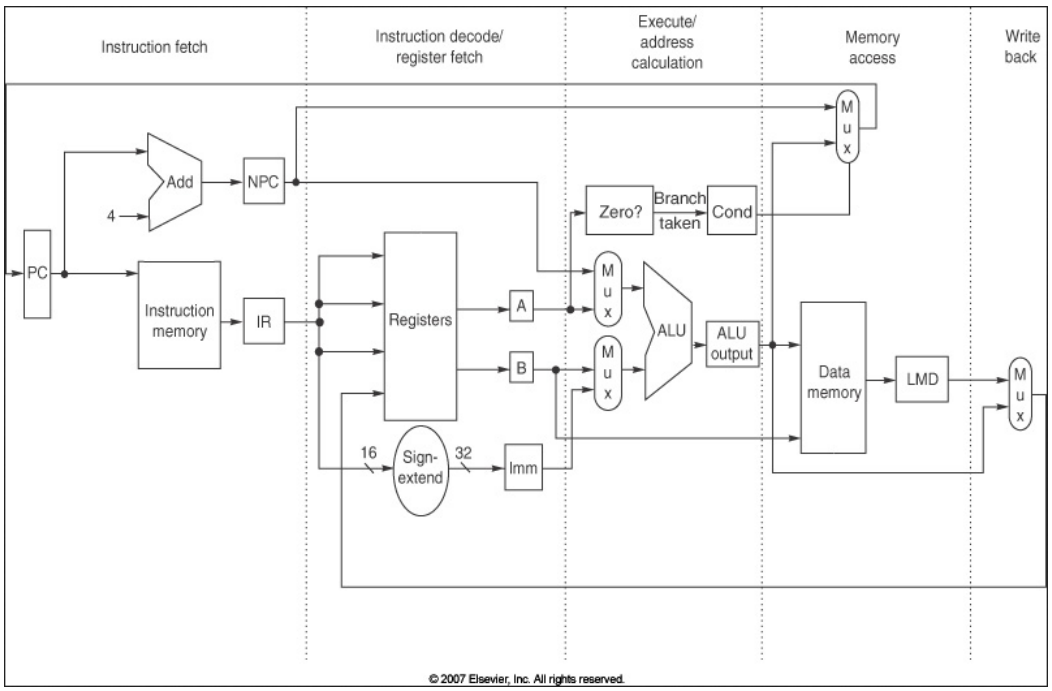

Figure 1.5: A simple breakdown of a generic processor architecture showing the typical division into five pipeline stages. Figure taken from [44].

1.6 Branch Prediction For Low Power Embedded Systems

Branch predictors in this field are generally different from high performance processor branch prediction units (BPUs) in that they tend to favour energy efficiency over cycle reduction. The major focus for designing a BPU for an embedded chip is to use as little space as possible, whilst making sure that the predictor is as accurate as possible. It is also important to ensure that as much energy is saved as possible - both in terms of extra cycles through mispredictions and through energy required to run the predictor itself.

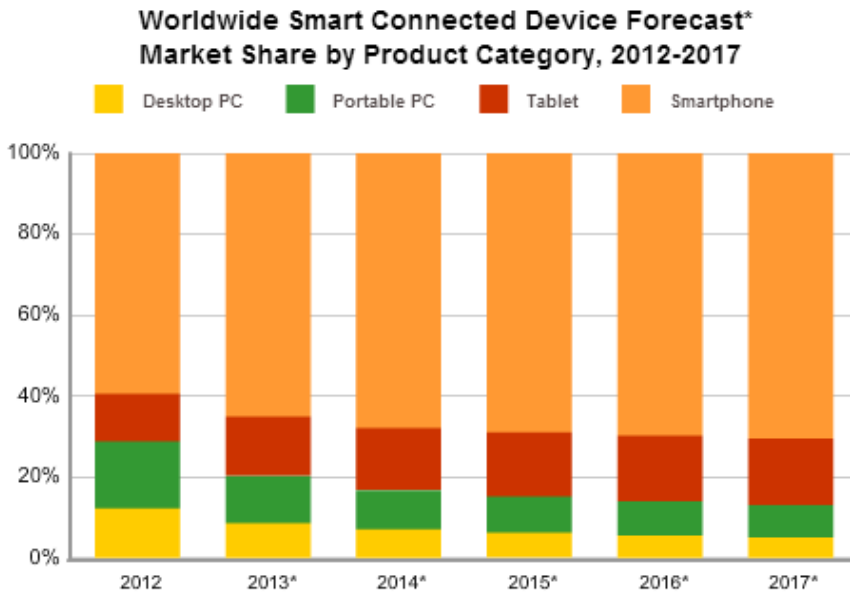

Figure 1.6: Many sources predict that embedded processors will be an increasingly important market over the coming years [35] [41]. Figure taken from [26].

The reasons for this are closely linked to the use of the devices. Embedded systems quite often draw their power from a battery. This implies that the power draw of the processor(s) is a large contributor to the battery life of the device. The size of the chip has an indirect effect on power levels, but also has a large impact on the price of the chip. This is because the manufacturing process relies on fitting many copies of the chip onto a single "wafer". This means the more chips that can fit onto a wafer the cheaper each chip will be.

This type of processor, where performance, price and battery life are each at a premium, are typical of the kind of processors found in modern mobile phones and tablet devices. These devices already form a large share of the market and their dominance is expected to increase over the coming years, such as seen in figure 1.6.

1.7 The Problem

1.7.1 Power

The power consumption of BPUs is an area of increasing concern. [49] makes an effort to quantify the energy used by a BPU and looks at the trade-offs based around energy use. It has been widely found that it is best to increase the energy consumption of a BPU in order to make it more accurate and thus reduce the number of cycles the

processor must run for, reducing overall energy consumption.

However, the increasing share of leakage energy and the rise of power hungry accurate processors means that this trend cannot continue indefinitely. There is always a balance to be struck between the power used and the power saved through time and accuracy gains. This balance is occasionally skewed in (battery powered) embedded processors, where it can be desirable to save power at the expense of performance.

Power considerations are most directly dealt with in a number of ways. Structures like the PPD [46] [49] are used to reduce the dynamic energy use of the predictor. Most predictors need to access the predictor structures every cycle in order to check whether the current instruction is a branch or jump in need of prediction. The PPD uses different hint mechanisms to detect when it is possible to skip the access to the branch prediction tables. Since every access to the predictor tables requires some energy use, reducing accesses will reduce overall dynamic energy used.

Power savings through reduced access can be achieved even more often through the use of sufficiently accurate static predictions [29] [81]. By using a static predictor rather than a dynamic predictor the energy that would be used in accessing the predictor tables is saved. However, static predictors are generally less accurate than dynamic predictors, so care must be taken in deciding which predictions can be made statically.

Similarly, papers such as [55] [68] [74] predict at what location in the BTAC the required branch will be in an attempt to reduce the active energy needed in the BTAC read through reducing the number of ways that need to be accessed.

A class of techniques called drowsy techniques are used to reduce the energy required to power the predictor tables [38] [59]. These work by putting all or part of the predictor into a low power "sleep" state. The disadvantage of this is that data cannot be read from the predictor in this state. It must first be returned to a high power state, taking extra time and delaying the prediction.

1.7.2 Aliasing

Aliasing within the BHT is a large problem, especially for smaller cache sizes [20]. Aliasing occurs when different branches with different outcomes try to access the same saturating counter. This results in the counter being both incremented and decremented, such that in the worse case the two conflicting branches never receive a correct prediction which they would receive if they were using two separate counters.

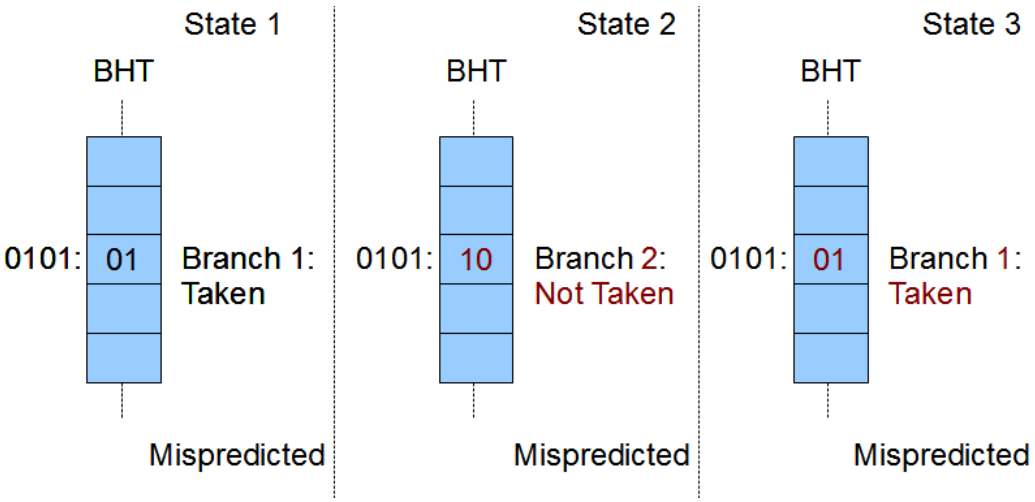

Figure 1.7: 2 branches both access the same 2 bit counter in the Branch History Table. The first branch is incorrectly predicted not taken and the counter updated to 10. The second branch is incorrectly predicted taken and updated to 01. The first branch then accesses the counter again and (because of the update from the second branch) is incorrectly predicted not taken.

Papers such as [53] look at reducing BHT aliasing by removing branches from the BHT through the use of static predictions. When the branches removed in this fashion are chosen correctly this can reduce active power consumption and aliasing within the predictor whilst helping make the most of the limited resources available within the prediction tables.

1.7.3 Capacity And Conflict Misses

Capacity and conflict misses are known in all cache-like structures and are of importance to BTAC structures in branch prediction [45]. These problems are less frequent than aliasing misses within the BHT but are more serious when they occur, as the presence of tags and the need to predict a precise target address means that there is no chance to 'happen upon' the correct outcome as may happen for BHT aliasing.

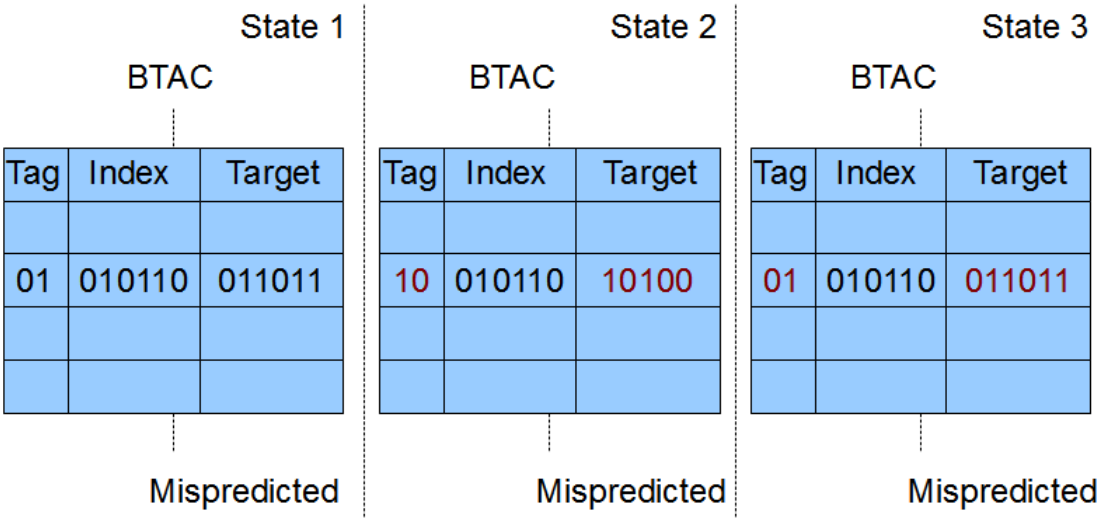

Figure 1.8: 2 branches both access the same entry in the Branch Target Address Cache. The first branch misses because this is the first time it has accessed the cache and so there is no predicted target. The second branch misses because a capacity miss results in it accessing the same entry and finding the wrong branch target. The first branch then suffers a capacity miss as its entry has now been replaced by the second branch.

There have been attempts to try and alter BTAC or BTB structures such that the branches that will be required next are pre-fetched into the BTAC [9] to try and avoid misses. The key insight here being that branches usually display strong temporal grouping, so that branches with a similar PC to the current PC are likely to be used in the near future. Papers such as [37] and [40] take a more active approach in trying to identify when and where BTB conflicts will occur and taking steps to avoid them, making use of new indexing functions and profiling information that informs the compiler of when branches can safely use the same BTB entries without risk of conflict.

1.7.4 Context Switches

The expectation is that with muticores and increasing speculation there will be an increase in short threads [14]. This will make it more important for BPUs to be able to maintain accuracy in the face of context switching. The problem with context switching is that, on a context switch, all data in the predictor become invalid (unless some of the BHT entries happen to be useful) and will be overwritten with new data.

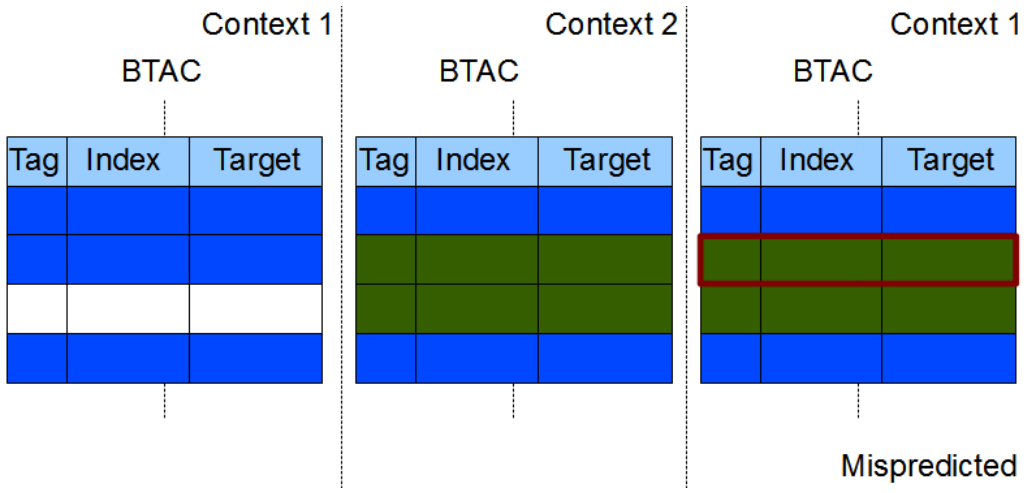

Figure 1.9: The BPU is first filled with branches from one context (in blue). Then there is a context switch and branches from the second context (in green) occupy the cache. This replaces some information from the first context. When the context switches back there is a misprediction when the first context tries to access the entry that was replaced by the second context.

This causes problems once the processor switches back to the original thread, as there is now a large number of conflict misses to contend with. Worse than this is the possibility that the BPU entries are invalid for the new thread, leading to erroneous branch target results, dirtying the BHT counters and introducing noise to the GHR.

Noise in the GHR is dealt with by [14] through setting the initial branch history to the PC of the branch as this was found to be almost as accurate as preserving the GHR and without the necessary extra hardware, time and energy for storage and retrieval. The approach taken in [51] is to store a compressed representation of the PHT, such as grouping entries by their bias bit or taking the average result across several entries. These compressed entries are then stored in an L2 cache or a dedicated buffer, the time taken for storage and retrieval is justified by the increase in accuracy that is obtained.

1.8 The Solution

In this section a high level overview of a pair of orthogonal solutions is presented. Each solution brings its own contribution to a predictor, eliminating a class of misses which was beyond the predictive power of the predictor previously. Each of the techniques can also be combined with each others and with many existing predictor types, to result in a new, more powerful branch prediction strategy.

1.8.1 Increasing The Use Of The Static Predictor In Dynamic/Static Hybrids

Predictors which combine the predictive power of a dynamic sub-predictor and the power savings found in a static sub-predictor are well known [80], [21]. However, there are additional savings to be made in the power use and die-space required for such a predictor.

The conventional wisdom is that only if a branch can be executed with the same accuracy on either sub-predictor is the branch moved to the static sub-predictor. This approach is modified by the addition of a parameter to allow for some accuracy to be sacrificed on the branch in question by executing it statically, in the hope of gaining energy savings and leaving more resources in the dynamic sub-predictor for the remaining dynamically predicted branches.

1.8.2 Sharing Information Between Cores

Data parallel applications are more and more commonly pushed towards GPU based solutions in an attempt to speed-up execution. However, GPUs are not well suited to applications rich in control flow instructions. In an attempt to make such applications better suited to execution on a multicore chip the knowledge that several copies of the same task are executed on multiple cores is leveraged for increased accuracy.

Messages are passed between cores containing information on recently retired branch instructions. This allows for the branch predictors to have accurate information with which to predict a branch, even if it is the first time the core encounters the branch. This behaviour leads to the cores producing the messages and the cores consuming them to cycle round, resulting in a raised average prediction accuracy.

1.9 Contributions

This thesis presents new techniques based on extensions to proven, existent branch prediction methods which maintain high levels of prediction accuracy while dealing with the unique opportunities and constraints seen.

- A novel static-dynamic hybrid mechanism which introduces a new design space parameter designed to allow a small amount of performance to be traded for energy and die-space savings.

- A possible mechanism for BPUs to communicate across a multicore system to increase the accuracy of each BPU.
- An exploration of how heterogeneous BPUs should be composed to achieve the most desirable performance to bits-budget ratio, both with and without communication between BPUs.

1.9.1 Outcomes

This thesis will be focused around trying to answer two major questions:

- Can BPUs for embedded processors be improved?
- How should existing BPUs technologies be combined to find the best BPU for a given chip?

The first question will be answered by going through the cutting edge technologies as described above and producing models to relate accuracy, die area and energy consumption in an attempt to be able to quantify the optimal points within the design space for a set of given design constraints.

The second question will be answered by producing a model for state space search and optimisation, which may prove useful in finding good points in the design space and attempting to answer how close to the optimal they are.

1.10 Structure Of The Thesis

The remainder of the thesis is organised as follows:

Chapter 2 sets out the difficulties in making accurate yet power efficient branch predictions in more detail.

Chapter 3 reviews other solutions to the problem of branch prediction.

Chapter 4 introduces the idea of pushing traditional dynamic-static hybrid predictors to make greater use of the static component to allow for smaller dynamic predictor components and resulting in die-space and energy savings.

Chapter 5 presents the novel idea of sharing information directly between branch prediction units, allowing new types of misprediction to be avoided.

Chapter 6 extends the work in chapter 5 by moving on to considering the advantages of heterogeneous branch predictors and how cooperative updates can be applied

to maximise the accuracy achieved by each BPU, whilst also keeping their size to a minimum.

Finally chapter 7 concludes the thesis; summarising the conclusions drawn from the work, re-stating the contributions made and discussing future work.

1.11 Summary

This chapter has introduced the unique problems facing low power embedded branch prediction, including power, aliasing, capacity and conflict misses, and context switches. It has advocated the use of a combination of novel techniques to increase the use of static predictors and share information between branch predictors.

Chapter 2

Background

2.1 Introduction

In chapter 1 the need for branch predictors was introduced. It was shown to be not sufficient to simply be an accurate predictor, but that the predictor must also meet the power, die-space and cost constraints which apply to the processor as a whole and depend on its application.

This chapter gives a short overview of the major branch predictor techniques, both static and dynamic, starting with the simpler techniques and building to more modern, accurate, complex predictors. For each predictor type that is introduced the key strengths and weaknesses of the technique are outlined.

Finally a summary of the different predictor types is given, followed by a description of which predictor types are used in chapters 4 - 6.

2.2 Overview

This section presents a brief overview of how the different predictor types compare. Table 2.1 shows how each predictor is characterised in terms of whether it makes use of static or dynamic prediction methods (or both), how high its prediction accuracy is, how high its energy consumption and die-space requirements are and how fast it can make a prediction.

Some of these properties will vary for the dynamic predictors depending on how large the predictor tables are. Table 2.2 gives a summary of the different parameters for each of the predictor types.

Technique	Static	Dynamic	Accuracy	Energy Used	Die-space Used	Prediction Speed
Taken	Yes	No	Very Low	None	None	Slow
BTFN	Yes	No	Low	None	None	Slow
Compiler Flags	Yes	No	Medium	None	None	Slow
Bimodal	No	Yes	Medium	Low-High	Low-High	Fast
Two Level	No	Yes	High	High	High	Fast
Hybrid	Yes	Yes	High-Very High	High-Very High	High-Very High	Fast
YAGS	No	Yes	High	High	High	Fast
COTTAGE	No	Yes	Very High	High-Very High	High-Very High	Fast
Neural	No	Yes	Low	High-Very High	High-Very High	Slow-Fast

Table 2.1: Summary of the different predictor types and their main features.

Technique	Prediction Structure Variables
Compiler Flags	Number of flag bits
Two Level	BTAC entries and associativity, RAS entries, BHT entries
Hybrid	Varies on sub-predictors. Additional BHT-like meta table/tables to decide between predictors.
YAGS	Tag length, entries and associativity for Taken and Not-taken caches
COTTAGE	Entries and tag width for base predictor and n different sub-predictors
Neural	Perceptron Weight table entries, history length

Table 2.2: Summary of the different parameters for each of the different predictor types

2.3 Static Branch Prediction

2.3.1 Hardware Based Prediction Mechanisms

The simplest solution to predicting a branch outcome is simply to predict that it is either always taken or always not taken. This can very easily result in low accuracy predictions. The approach was rapidly improved by the introduction of Backward Taken Forward Not-taken (BTFN) prediction schemes. The key insight here being that backwards branches, such as loop ending branches, are generally taken, where as forward branches are often not taken.

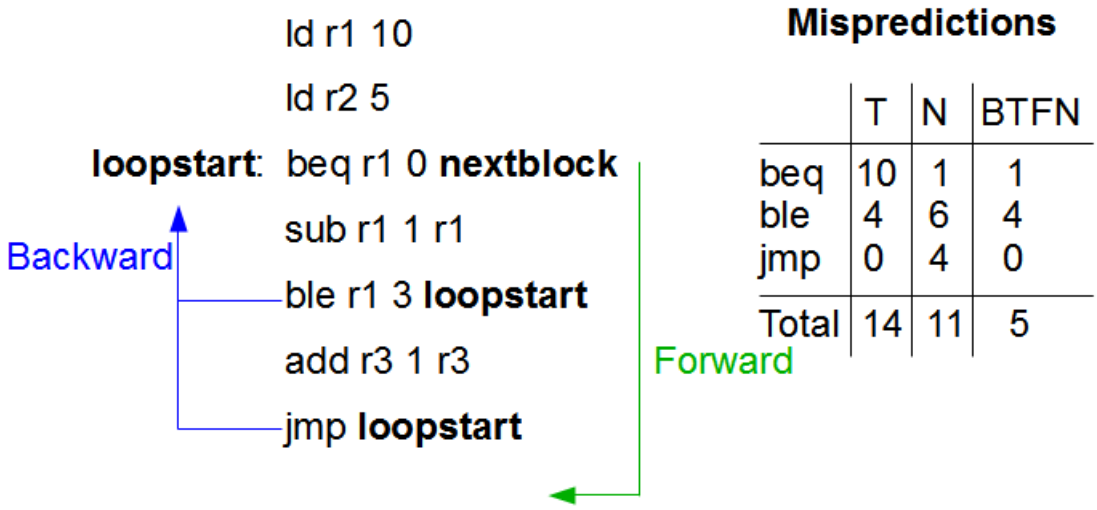

Figure 2.1: Comparing the always taken (T), always not taken (N) and Backwards Taken Forwards Not taken (BTFN) static prediction schemes.

2.3.2 Static Branch Hints

BTFN has been proven a reasonable general strategy [64], especially considering the little resource required to implement it. However, it is possible to encounter pathological examples where simply reversing the prediction direction for a given branch results in increased prediction accuracy. Such an approach can be easily taken through use of a flag bit in the branch instruction, which is set in the compilation of the application.

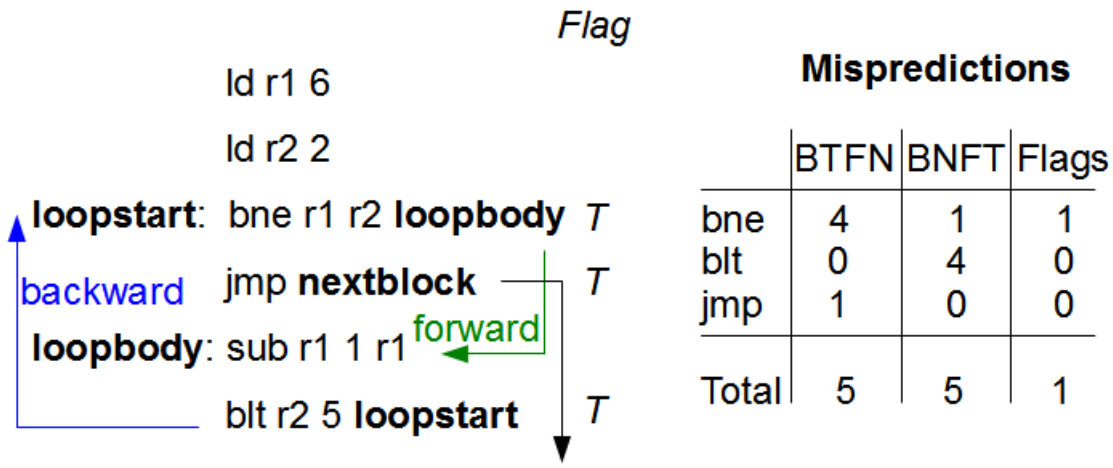

Figure 2.2: Comparing the BTFN, BNFT and compiler flag prediction schemes.

In [3] the idea of forming a prediction for a branch based on a number of properties of that branch was introduced. A number of interesting properties of branch instructions were defined, each with an associated probability of the branch being taken or not taken. By combining together the probabilities applicable to any given instruction a likelihood of the branch being taken is produced. This is then used to set a predict taken flag bit in the instruction.

An alternative method to this is the use a simulator to conduct a profiling run of how the program executes. This provides information on how often a branch is predicted at run time and how often it is taken or not taken. The advantage of this is that there may be branches which exhibit behaviour different to that predicted by a compiler based mechanism.

One disadvantage of profile based hints is that it takes extra time to conduct the profiling runs, however this is a one off cost that can dramatically increase the predictor accuracy. A more serious disadvantage is that the behaviour of a branch may depend on the input dataset. If this is the case it may be possible for the branch to act entirely opposite to the profiled hint. While this may be no worse than performance arising from a bad compiler based hint, it is now at the extra cost of the wasted profile run.

Chapter 4 makes use of a profiled branch hint mechanism to set a flag bit in the branch instruction. This hint bit is used to flip the standard BTFN prediction to a BNFT prediction if the bit is set to 1.

2.4 Dynamic Predictor Types

The simplest dynamic branch predictor method is the 1-bit counter. The counter takes the value of the last branch outcome (1 for taken, 0 for not taken). This value is then used to predict the next branch outcome. This prediction mechanism suffers from single instances of deviant behaviour from an otherwise regular pattern, such as that seen at the end of a loop. As a result the approach was generalised to the bimodal predictor.

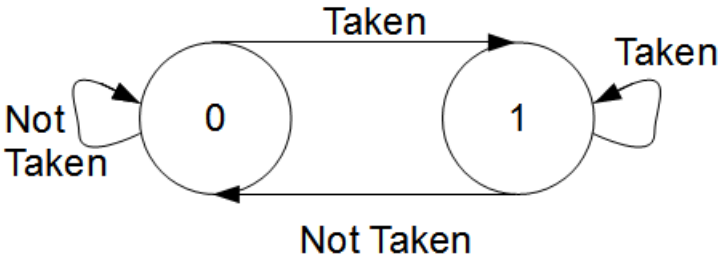

Figure 2.3: When the counter has value 0 predict not taken. When it has value 1 predict taken

2.4.1 Bimodal

A bimodal predictor is amongst the simplest of dynamic predictor mechanisms. It comprises of an n-bit saturating counter, with the highest bit used to govern the prediction. When the most significant bit is 1 the branch is predicted taken, when it is 0 the branch is predicted not taken. Every time the branch is taken the counter is incremented. Every time the branch is not taken the counter is decremented.

The method of relating a branch to a counter is very important and differs from scheme to scheme. A bimodal predictor is generally comprised of 2 bit counters and is indexed by the branch PC. Ideally there should be a separate counter for each branch. This has been shown to result in accuracy as high as 93% [23].

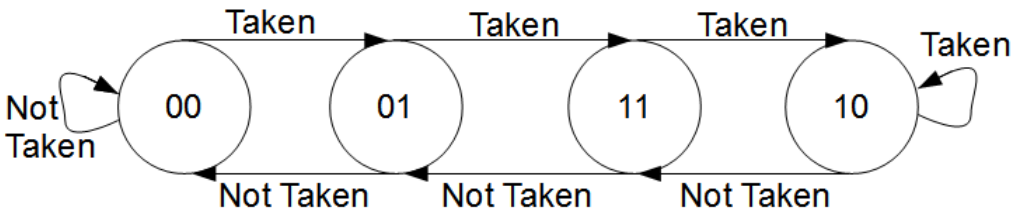

Figure 2.4: When the counter has value 00 or 01 predict not taken. When it has value 10 or 11 predict taken

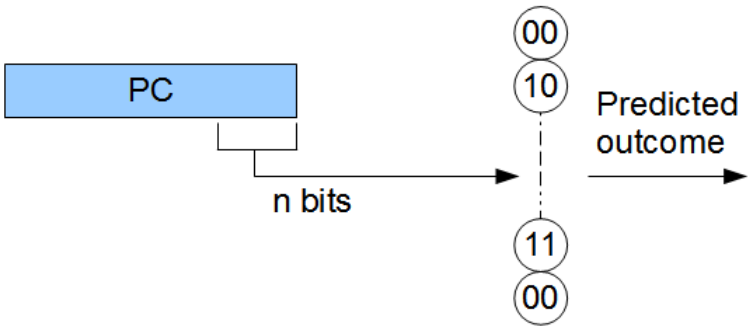

Figure 2.5: A bimodal predictor. Using the lower n bits of the PC allows for 2^n counters to be addressed. If n is not high enough then more than one PC can be mapped to the same counter.

2.4.2 Two Level Predictors

It has been observed that there are many instances where the outcome of one branch can directly affect the outcome of another branch (e.g. a series of if statements checking a variable against a series of values). Two level predictors capitalise on this information by using the branch history (the outcome of the previous n branches) to index a Pattern History Table (PHT), containing 2-bit saturating predictors such as found in a bimodal predictor. This leads to highly accurate predictors which can give a correct prediction provided that the history length is long enough and that there are no two instances of the branch history having the same value but the next prediction having different outcomes.

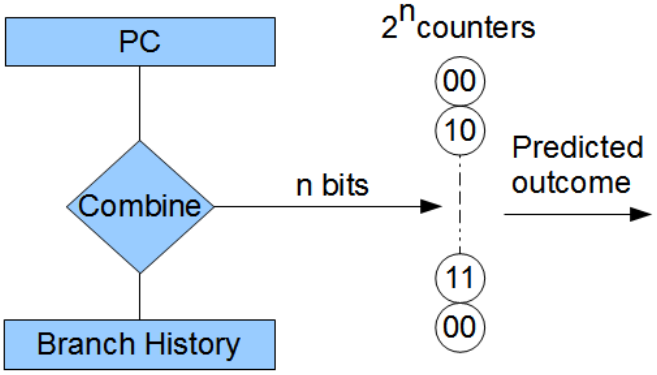

Figure 2.6: A global two level predictor. Some combination of n bits of the PC and branch history are used to address the 2^n counters. If n is not high enough and the history bits have certain values then more than one PC can be mapped to the same counter.

Two level predictors are classified as either local or global depending on how the branch history is collected. Local predictors use a separate branch history for each branch and may also have a separate PHT for each branch. Global predictors share a single branch history (and PHT) between all branches.

2.4.3 GShare/GSelect

GShare and GSelect are two of the most popular variants of global two level branch predictors. A GShare predictor works by XORing the lower n bits of the branch PC with the global branch history to produce the index to the PHT. A GSelect predictor works by concatenating some part of the branch PC with the global branch history.

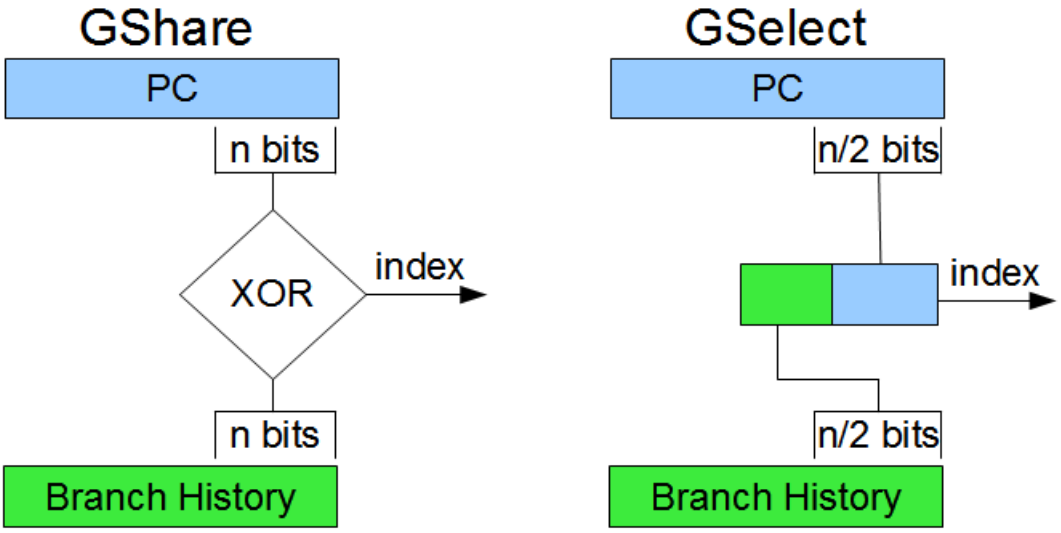

Figure 2.7: The difference in indexing methods between GShare and GSelect. Both are global two level predictors, using the index produced to access a table of 2-bit saturating counters. GShare XORs together n bits from both the history and the PC. GSelect takes n/2 bits from the PC and branch history, concatenating them to form the n bit index.

GShare and GSelect predictors both capitalise on and suffer from the effects of aliasing, one of the major problems facing branch prediction as introduced in section 1.7. Aliasing occurs when two different branches access the same 2-bit saturating counter in the PHT. This can have either positive or negative effects. Positive aliasing occurs when the two branches have the same outcome, resulting in the second branch accessing a counter that has been through its training period and is carrying the right value to predict the new branch. Negative aliasing occurs when two branches accessing the same counter have opposite outcomes. In the worse case this can result in a counter

flip flopping backwards and forwards between taken and not-taken predictions. This results in two branches that could be trivially correctly predicted both being incorrectly predicted every time.

Chapters 4, 5 and 6 make use of a GShare predictor. This is because of the high accuracy and low complexity afforded by a GShare predictor. This makes them easy to implement in both hardware and software and facilitates investigation into the effects of new prediction mechanisms without the complexity of the predictor making the results harder to evaluate.

2.5 Hybrid Predictors

Hybrid branch predictors come in many different varieties and can be broadly classified based on which types of sub-predictors they include. The simplest type of hybrid predictor is made up of two sub-predictor components, such as those introduced in section 1.4 which often created by using GShare and BTFN sub-predictors. Branches that are trivially predictable are predicted with the static predictor to avoid the higher power requirements of predicting with the dynamic predictor. Branches that are very hard to predict are also predicted with the static predictor to avoid the high penalty of the dynamic predictor mispredicting. The remaining branches are dynamically predicted to achieve better accuracy than is possible with the static predictor. The decision of which sub-predictor to use is either taken from small cache structures accessed before the main branch predictor, such as the Prediction Probe Detector (PPD), or by dedicated hint bits in the instruction representation.

2.5.1 Controlling Sub-Predictor Accesses

The PPD (further explored in section 3.2.2) is a much smaller cache than the main branch prediction caches. It is accessed before any other branch prediction caches and functions in a very similar to way to the PHT. If the access misses then the processor continues as if the PHT had been accessed and no entry had been found. On a hit the PPD entry is used to select which sub-predictor to use to make the prediction, either the static or dynamic component. The key concept in this case is that the small size of the PPD allows for quick access, thus not lengthening the pipeline stage and reducing the dynamic and static energy use as well as the die area required.

The hint bits approach is reliant on there being enough redundant bits in all instruc-

tions for the re-tasking of the bits as branch hints, or the addition of additional bits to the instruction representation for ISAs with variable instruction lengths. This approach avoids any new cache structures being added to the processor, but is reliant on the extra control flow logic to be able to quickly extract the relevant bits and then access the branch predictor structures as needed. This requires a sufficiently long pipeline stage for accessing both the I-cache and BPU in series in one cycle. The combination of the ISA and clock frequency requirements are very restrictive in many circumstances, making the PPD approach preferable where the energy and die space can be spared.

By ensuring that the dynamic and static sub-predictors are accessed only for the branch instructions best suited to their prediction capabilities, this type of hybrid predictor can address the issues of aliasing and power consumption, two of the major problems facing branch prediction as identified in section 1.7. This allows for the creation of a highly accurate predictor which is also die-space and energy efficient.

2.5.2 Other Hybrid Predictor Types

A different kind of hybrid predictor is constructed when each of the sub-predictors is accessed on every branch prediction. These sub-predictions are then selected from or combined in some manner. A simple approach is to take the majority vote from the sub-predictors as the final prediction. An alternative approach is to introduce a cache structure to form a meta-prediction table. This stores dynamically updated counters which are indexed by the branch PC and used to select which sub-predictor provides the final prediction. While this approach does require the addition of a new cache structure it can typically achieve higher rates of accuracy by ensuring that predictor with the correct resources to predict the branch gets to make the final prediction. This scheme can even be extended by accessing the meta-predictor first and then only accessing the selected sub-predictor for a branch outcome prediction, however this requires that the meta-predictor and any of the sub-predictors can be accessed in sequence in a single clock cycle.

A further approach to constructing hybrid predictors is to combine predictors which provide predictions at different speeds and different levels of accuracy. Perceptron based branch predictors, discussed in detail in section 2.6.2, provide highly accuracy predictions but can take many cycles to do so. As discussed in chapter 1, the time taken for a prediction to be made has an impact on program execution time and so on the total energy required. As a result perceptron predictors are sometimes used in

a hybrid predictor where the other sub-predictors are more traditional designs, such as a GShare predictor, which can produce a prediction in a single cycle. This quick prediction is used by the processor to progress the application while the perceptron predictor is produced. If it is found that the perceptron prediction disagrees with the earlier prediction then the pipeline is flushed in much the same way as a standard branch misprediction, except that the perceptron prediction becomes available before a standard branch prediction could be detected.

2.5.3 Thesis Contribution

Section 2.5 explained how hybrid predictors such as a static-dynamic predictor can be used to produce a predictor that is greater than the sum of its parts. The static predictor with branch hints and GShare predictor highlighted in this chapter are used to form the basis of the techniques in chapter 4. We improve upon the basic design with a new consideration for which sub-predictor should be used for each branch. By predicting more branches statically we trade a slight execution slowdown for a larger energy saving.

This also reduces pressure on the dynamic predictor, resulting in less aliasing occurring. This can in turn result in a smaller dynamic predictor achieving the same accuracy rates as a larger predictor that does not use the technique, allowing for further energy, die-space and cost savings.

2.5.4 YAGS

The YAGS Branch Prediction Scheme [20] is a key example of attempts to use a new predictor architecture to address aliasing within the BPU. The scheme reducing aliasing in the PHT through extending the ideas of the agree and bi-mode predictors of splitting the PHT into taken and not-taken sections. The predictor uses a bimodal sub-predictor to store branches grouped by bias, and then uses the PHT to predict instances where the branch will deviate from its bias. This massively reduces aliasing in the PHT and allows for a smaller PHT to be used to offset the size of the hybrid. The predictor in [59] takes a similar idea of predicting when a branch will deviate from its normal outcome and uses a branch mispredict predictor to overturn the branch predictor predictions, but this time the prediction is based on the number of committed branches since the last misprediction.

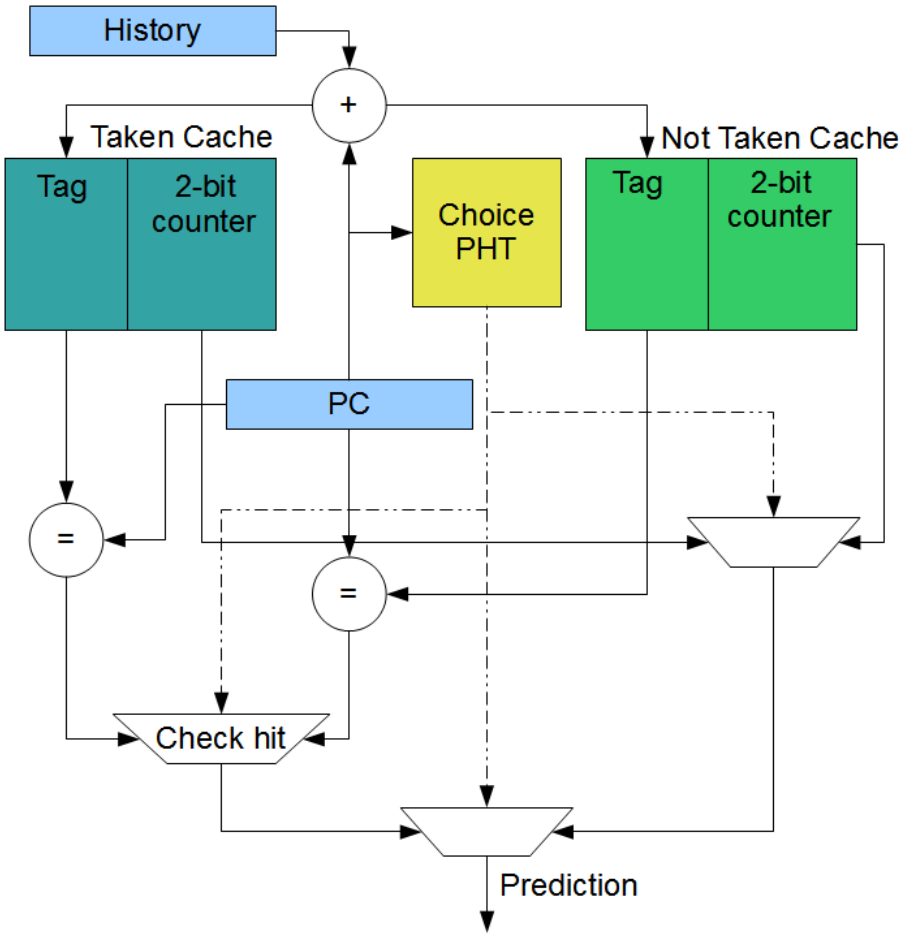

Figure 2.8: The YAGS predictor based on the original diagram from [20]. The choice predictor is accessed to check the bias of the branch and the cache for the opposite outcome is checked for a special case where the branch does not agree with the bias. If a special case is found its predicted outcome is used, otherwise the bias outcome is used. To reduce aliasing, small tags are added to the PHT and used to form a set associative cache. This reduces conflict misses that cause aliasing.

The problem with the YAGS predictor is that the number of tables involved increases the complexity of the design and thus the hardware implementation. The extra complexity of the hardware implementation means that it may well require more die-space than the equivalent sized GShare or GSelect predictor, however [20] suggests that it will be more accurate.

A second problem is that the extra tables results in an increased number of lookups. Each table lookup requires an amount of dynamic energy to read the stored data and check the tag bits. Thus, the YAGS predictor may well require more energy than a simpler branch prediction scheme.

2.6 Alternate Dynamic Predictor Types

State of the art predictors do not always follow the template of using several varieties of two-level predictors combined into one larger hybrid predictor, with possible additional sub-predictors. The following are examples of highly accurate predictors based on alternative prediction methods.

2.6.1 COTTAGE

The COTTAGE predictor is the result of the development of the TAGE and ITTAGE predictors [62]. The use of partially tagged predictor components, each using a different history length comprising a geometric series, allows for branches to be predicted using a level of resources sufficiently large without being wasteful. This comes from the observation that different branches require different history lengths (or even history types, either local or global) to be predicted with highest accuracy.

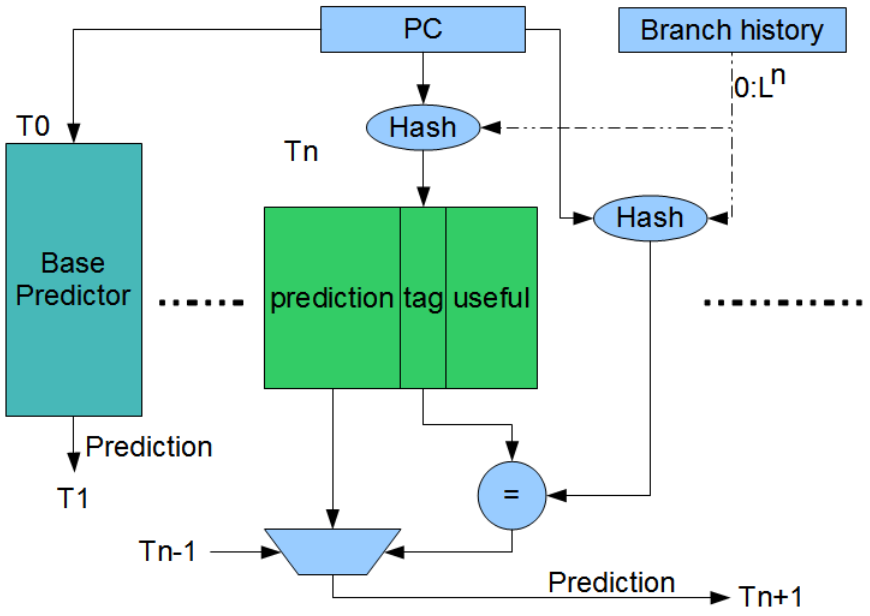

Figure 2.9: The TAGE predictor based on the diagram from [62]. A number of different sub-predictors are used, each with a different history length. The history length used for each sub-predictor forms a geometric series. In this way the different branch types which require different history lengths to be correctly predicted are all provided with exactly the resources they need. The useful counter is used as part of the predictor update process. The ITTAGE and COTTAGE predictors build on the same basic structure.

The problem with the COTTAGE predictor is similar to that faced with the YAGS predictor. The large number of components means a large complexity to the hardware. The large number of components that must be accessed also leads to a high dynamic energy requirement due to the large number of table lookups. A second problem is that the geometric history length progression results in a very large history length that must be tracked for the larger predictor components. This results in some long tags and complex hardware needed to cope with them, further increasing the energy, complexity and die-space requirements of the predictor.

2.6.2 Neural branch prediction

This class of predictors replaces 2-bit saturating counters with techniques such as multilayer perceptrons [1] [24] [25] [32] [60] [69]. These predictors are able to exploit very long history lengths, while requiring less resource than would be required in a two level predictor, resulting in increased predictor accuracy. The main problem encountered with these predictors is that they take a long time to access, calculate the prediction and return the predicted outcome.

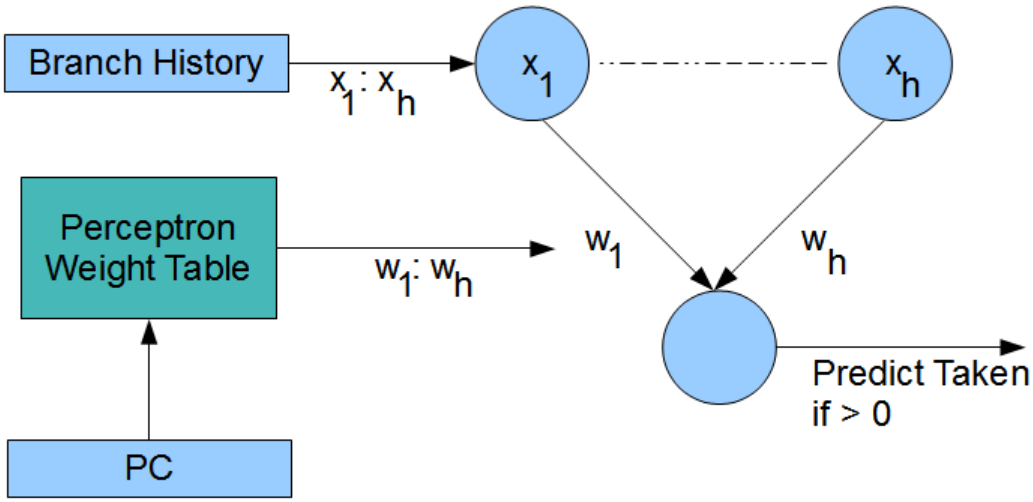

Figure 2.10: A perceptron branch predictor partially based on the diagrams from [32]. The PC is used to select the perceptron input weights. The inputs are the last h branch outcomes. Generally a much longer history length is used than with traditional predictors. In this way the branch outcome is correlated with the outcomes of each of the last h branches. Each past taken branch is stored as a 1 and each not taken branch as a -1. Each input is multiplied with its associated weight and then the sum of all these values is taken. If the result is greater than 0 then the branch is predicted taken.

It has been shown [31] that in modern, aggressively pipelined processors the cost of a prediction taking more than one cycle is often more expensive than a less accurate predictor that returns a result in one cycle. This is as a result of the stalls introduced when waiting on easy to predict predictions. This prediction latency has been addressed in two ways. The first is to use a second predictor (such as a small two level predictor) which can deliver an accurate prediction quickly. This first prediction can then be overridden by the more accurate prediction from the neural prediction if required. Alternatively the access latency is reduced by starting the access head of time, using partial or path based information which is refined by the time of prediction.

2.7 Summary

This chapter has introduced a variety of different branch prediction techniques. The classification of static or dynamic branch prediction techniques was introduced, along with a brief summary of the major developments for each class of predictor. Each of these developments was presented with the problems they solve and the limitations they suffer from.

Static branch prediction techniques were shown to be a simple yet powerful tool for predicting highly predictable branches in a simple, energy efficient and low die area manner. The development of compiler hints and other profile based static prediction greatly increased accuracy.

Dynamic branch prediction methods are able to capture far more complex patterns in branch behaviour, especially with more recent techniques that can capture a very large branch history. Dynamic predictors require more hardware resources than static predictors, resulting in increased die area requirements, increased dynamic power and increased leakage power.

The analysis of the branch prediction methods introduced here has shown that modern branch predictors have become much more accurate through addressing the problem of aliasing. However, they have often done so at the expense of greater power and energy requirements. This thesis aims to build on these techniques to provide a new branch prediction method that retains the high accuracies achieved yet reduces the power and energy requirements.

The static predictor with branch hints and GShare predictor highlighted in this chapter are used to form the basis of the techniques in chapter 4. This allows for the creation of a highly accurate predictor which is also die-space and energy efficient. The

combination of the two predictor types allows for branches with low static accuracy to be dynamically predicted and for dynamically aliasing branches to be predicted statically.

Chapter 3 introduces some of the most recent research into new variants of branch predictors. The techniques found in this chapter and in chapter 3 will be built upon and added to in the implementation of the novel techniques introduced in chapters 4 and 5.

Chapter 3

Related Work

3.1 Introduction

This chapter presents a review of work on branch predictors that have attempted to tackle the same type of problems as targeted by this thesis. They are the most up to date, accurate and efficient techniques available.

The first series of papers considers the problem of power consumption identified in section 1.7. The approach taken in *Branch Prediction On Demand: an Energy-Efficient Solution* [13] looks at dynamically managing the portion of the predictor that is active and changing it to meet the needs of the program. The approach taken in *SEPAS: A Highly Accurate Energy-Efficient Branch Predictor* [5] reduces dynamic energy consumption through detecting when a predictor has reached a steady state (where updates have no impact) and prevents further updates. Finally, *A Break-Even Formulation for Evaluating Branch Predictor Energy Efficiency* [15] highlights the importance of the balance between the accuracy of the BPU, the energy consumed by the BPU and the impact the BPU has on application runtime (and therefore energy usage). Between them these three papers serve to highlight the importance of the energy consumed by a BPU and ways to address its dynamic and static energy consumption.

The next series of papers also addresses the problem of power reduction, by targeting power consumption itself rather than energy consumption. The results presented in *Power-Aware Branch Prediction: Characterization and Design* [50] are a collection of different techniques (some from cited previous papers) that can be used to reduce the dynamic and static power consumption of the BPU. The technique presented in *Power Efficient Branch Prediction through Early Identification of Branch Addresses* [76] goes one step further by transferring the low power drowsy state previously seen in I-cache

and D-cache designs to the BPU.

The problem of power consumption is also addressed in the next series of papers, this time through the use of compiler based approaches. The approach taken in *Power-aware branch prediction techniques: a compiler-hints based approach for VLIW processors* [46], focuses on exploiting a VLIW architecture to produce highly efficient branch hints, resulting in a dramatic reduction in BPU accesses and processor energy consumption. The paper *Towards an Energy Efficient Branch Prediction Scheme Using Profiling, Adaptive Bias Measurement and Delay Region Scheduling* [3] presents a more generalised approach to using branch hints at run-time to determine which parts of the BPU need accessing. This increases accuracy and eliminates unnecessary lookups, thereby reducing runtime and BPU dynamic energy consumption. Finally, *Energy-Efficient Branch Prediction with Compiler-Guided History Stack* [67] takes a different approach, modifying the instruction stream to insert entirely new instructions addressing the BPU directly. This approach helps to reduce capacity and conflict misses, resulting in increased accuracy and reduced energy consumption. These papers highlight the power and flexibility that the compiler can bring to the challenge of run-time branch prediction, providing extra information to aid the dynamic predictors in their ability to accurately predict branches.

In *Branch Classification: A New Mechanism for Improving Branch Predictor Performance* [11] the effectiveness of compiler based branch prediction hints is expanded through increasing the selection of predictors that the hints can target a branch at and the specialisation of predictors for the selected type of branch. The paper *Combining Static and Dynamic Branch Prediction to Reduce Destructive Aliasing*[53] presents a technique that addresses both capacity/conflict misses and energy consumption. This is achieved through the replacing of accesses to the dynamic predictor with static predictions, reducing the dynamic energy used to access the dynamic predictor and reducing the pressure on the dynamic predictor entries. Energy improvements can also be leveraged through using different types of dynamic predictors, such as shown in *Low Power/Area Branch Prediction Using Complementary Branch Predictors* [59].

The remaining papers take a closer interest in solving problems to do with multi-threaded and multicore applications and hardware, aiming to increase predictor accuracy and thus reduce runtime and energy consumed. A novel processor architecture is presented in *A Study of Slipstream Processors* [58], with two cores working in tandem to identify the instructions critical to the execution of a program, discarding those that are less important and further speeding up execution through branch hints from

a run ahead thread. The problem of maintaining accuracy under context switching is addressed in *Accurate branch prediction for short threads* [14], through the priming of the GHR to help give meaningful branch history information. The approach taken in *The research of Multi-Core architecture's predictor* [34] is to introduce a new shared predictor table, leveraging the information from multiple predictors to increase accuracy. Finally, in *How to Implement Effective Prediction and Forwarding for Fusible Dynamic Multicore Architectures* [57] a further approach to novel architecture is presented, with a technique that fuses and splits cores as dictated by the requirements of the application at runtime.

	Energy	Power	Hints	Hybrid	Cooperation
1994				9 [11]	
1995					
1996					
1997					
1998					
1999					
2000				10[53]	12[58]
2001					
2002					
2003	1[13]				
2004	2[5]	4[50]	6[46]		
2005	3[15]				
2006		5[76]			
2007			7[3]		
2008				11[59]	13[14]
2009					
2010					14[34]
2011					
2012			8[67]		
2013					15[57]
2014					

Table 3.1: Year of publication and bibliography reference for the papers featured in this chapter. Papers are grouped by subject into columns. The numbering refers to the order in which the papers are discussed in this chapter.

3.2 Efficiency

Efficiency can be achieved through various means, which are focused on various goals. Here we present papers organised by the methods they using to achieve the desired efficiency.

3.2.1 Energy Efficiency

More recent papers have started to look at the important role that the branch predictor has to play in the energy consumption of the processor as a whole. While some (usually older) papers look at the branch predictor in isolation [1] [12] [27] [30] [36] [39] [40] [54] [76] [82] , more recently there has been a growing focus on the impact the branch predictor can have on the power use of the entire processor [5] [8] [13] [16] [22] [29] [42] [46] [47] [49] [50] [59] [75] [83].

1. Branch Prediction On Demand: an Energy-Efficient Solution, 2003 In [13] we find an approach which takes action at runtime to reduce the energy consumption of the BPU. The BPU considered in the paper consists of a hybrid predictor made of Gskew, Pskew and bimodal components. To find the best BPU layout the program is split into smaller sections which are then profiled offline. Based on the profiling results, access to the Gskew and Pskew components can be dynamically disabled.

It is noted that the requirements on the BTB can vary wildly with the application, some consisting of many more static branch instructions[1] than others. As a result the BTB can also be dynamically resized based on the profiling run through reducing either the number of active sets (from 2048 to 256) or the number of active ways (from 2 to 1).

The results presented show that an average reduction of 71.7% of branch predictor energy and an average energy reduction of 6.2% across the processor as a whole.

Comment This paper takes an interesting approach to the problem of power consumption and its efforts to maximise the efficiency of the predictors make a large impact in the energy requirement of the BPU. However, there are a few drawbacks to this approach. At the end of section 4.3, the paper notes that the technique is not only dependent on the efficiency of the implementation, but upon the demands of the branches

[1] This refers to the number of different instructions found in the code, rather than the number of branch instances encountered at run time

encountered in the code. As a result it is possible that the technique could be much less effective on certain benchmarks. This is because it might be possible for none of the BPU configurations available at runtime to be ideal for the benchmark.

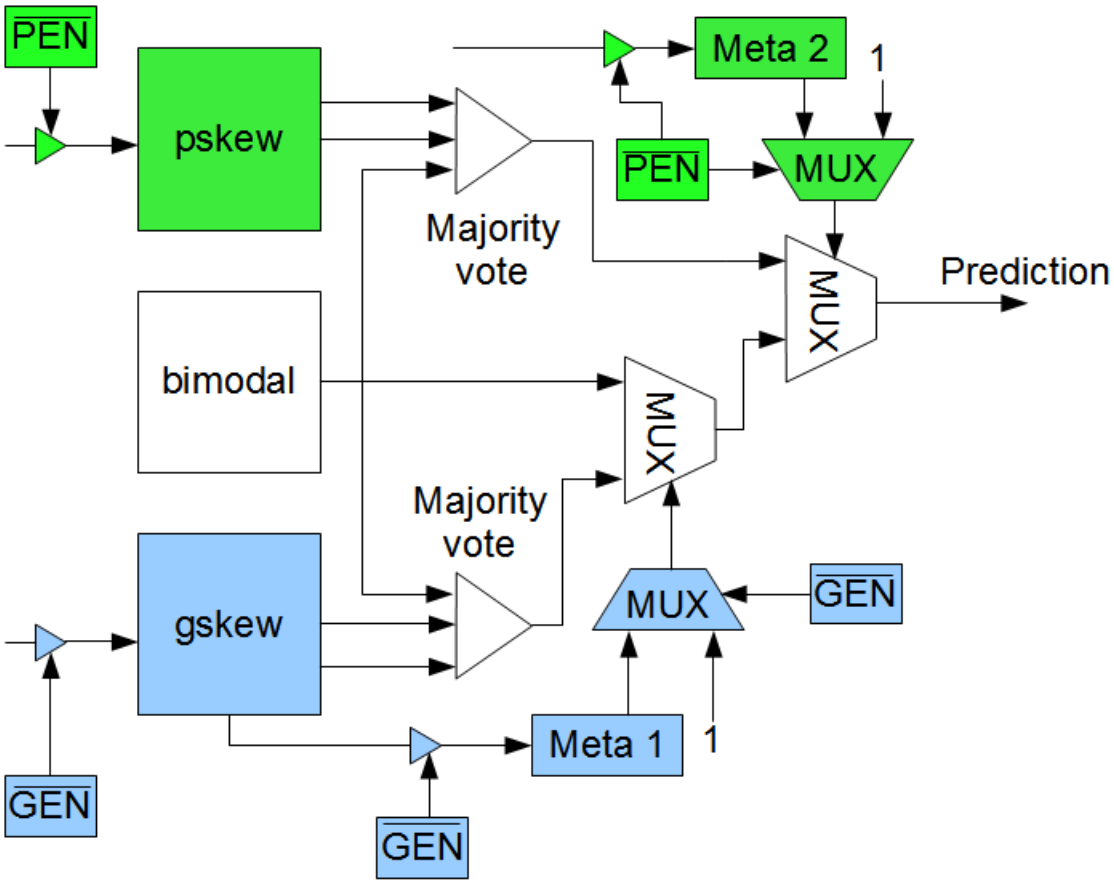

Figure 3.1: The 2Bc-gskew-pskew predictor used in [13], based on a figure from that paper. The GEN signal is used to gate access to the gskew and Meta 1 tables, while the PEN signal is used to gate access to the pskew and Meta 2 tables respectively.

If the hardware is not properly tuned to the software, or if the range of dynamic reconfigurability is not sufficiently large and flexible then performance will suffer. In the best case, the profiling run will show that a single configuration is highly suitable. This would make the offline profiling a wasted effort as the benchmark would perform well without the technique. Similarly, if the profiling shows that none of the available configurations is suitable for the benchmark then the profiling is wasted as the dynamic reconfigurability would be unable to provide a sufficiently accurate BPU. Alternatively, the profiling may show that the range of benchmarks requires highly similar BPU configurations, in which case the dynamic adaptability would have limited usefulness.

Each of these cases shows the importance for the range and flexibility of the available hardware reconfigurations to match the software likely to be run. Otherwise there would be a large overhead in the wasted hardware provided for making the BTB sets, BTB ways and direction subpredictor accesses reconfigurable.

It is also possible to consider a situation where the technique may be 'too effective', taking structures that are not appropriate and altering them to be more so when a better solution would be a different base predictor. In such a 'best' case scenario this technique results in large amounts of the BPU being inactive, resulting in effectively wasted die space and an unnecessarily costly and complex BPU. It would be more desirable to have a predictor where all parts of it can be usefully employed at all times. The problem then returns to the core of branch prediction: what would the ideal predictor look like? Until this is answered the technique presented here seems very reasonable.

2. SEPAS: A Highly Accurate Energy-Efficient Branch Predictor, 2004 It has been noted that branch predictor tables generally go through a warm up phase before settling into a steady state phase. This steady state phase is typified by the branches showing stable, easily predictable behaviours. This behaviour is used in [5] where accesses and updates to branch predictor components are removed though the use of a new meta predictor, the SEPAS filter table. The technique can dramatically reduce the number of predictor lookups and updates, while only reducing performance by a maximum of 0.25%.

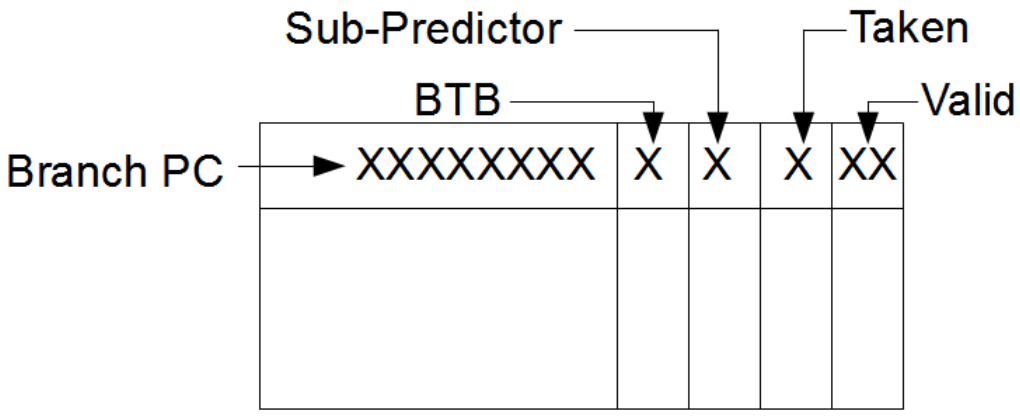

Figure 3.2: The SEPAS filter table, figure expanded from [5]. Branch PC holds full address of the branch instruction. The BTB, Sub-Predictor and Taken fields are a single bit each. The Valid field is two bits.

The SEPAS filter table used in [5], shown in figure 3.2, was a 256-entry direct-mapped cache. This was used for a selection of hybrid GShare and Bi-modal predictors ranging from a 32k-entry combined predictor with 1k, 4-way entry BTB to an 8k-entry combined predictor with a 256, 4-way entry BTB. This means that the SEPAS filter table achieves a very low overhead, especially for the larger BPU. SEPAS filter table entries consist of PC, BTB (1-bit, records if entry is in BTB), sub-pred (1-bit, records last sub-predictor used to predict branch), taken (2-bit saturating counter used to count how many times branch was taken successively) and valid (1-bit, used to select if access to other predictors should be avoided) fields. A branch is defined to be in the steady state if it has been taken three times in a row, and was correctly predicted (not-taken branches are not considered).

The paper does not make use of a specialised loop predictor, instead the SEPAS filter is used to filter out loop branch predictions. This means that there will quite likely be a large number of loop entries in the SEPAS filter. It would be interesting to see an analysis of how using both and loop predictor and a SEPAS filter would impact on the required size of both structures and what effect this would have on energy consumed.

Comment While the technique will significantly reduce dynamic energy through the elimination of unnecessary accesses and updates, it will have no impact on the static energy requirements. It would be more desirable to see some attempt to introduce techniques similar to those found in [13] to set parts of the predictor to a low power state, thus reducing static power consumption. Furthermore, the technique requires a (small) increase in die space. Given that the disabled accesses reduce pressure on the main predictor it may be possible to reduce it in size (not simply power gating but smaller predictor tables), as long as the steady state is a large enough portion of the program and the reduction in resources does not reduce accuracy too much in the training phase.

3. A Break-Even Formulation for Evaluating Branch Predictor Energy Efficiency, 2005 It is noted in [15] that the energy budget of a branch predictor can be a significant portion of the energy budget of the processor as a whole, up to 11%. The paper presents a number of conclusions on what they term the “break-even” energy point of the predictor, the point at which the energy used by the predictor is equal to the energy saved by its correct predictions. This is done through the formalisation of several equations used to represent the ED^2 processor, the branch predictor and their

sub-components.

The paper presents the result that the energy impact of a branch predictor is independent of the pipeline width and cache size, meaning that when new branch predictor designs are evaluated it is possible to focus solely on the predictor in isolation. This is then combined with the result that an increase in predictor energy use often saves power across the chip as a whole as a result of the extra branch prediction accuracy (and thus mispredicted cycles that are avoided).

Comment The framework produced in [15] demonstrates that it is important for branch predictors to not simply be accurate but energy efficient. Whereas most papers have made an attempt to improve upon the break-even formulation necessary for the BPU through increased accuracy, this thesis instead aims to make this improvement through reducing the energy use while keeping the accuracy the same.

3.2.2 Power Reduction

While very closely linked to energy efficiency there are some slight differences when optimising for power reduction. Power reduction is important for embedded devices, where there is a hard limit on the maximum power available at any time. Reducing average power consumption will reduce energy consumption.

4. Power-Aware Branch Prediction: Characterization and Design, 2004 The portion of the predictor used to predict the direction of the branch consumes less than 1% of the entire processor power, while the BPU as a whole may often consume 7-10% of the processor power budget. [50] presents the view that it is better to spend more power on the direction predictor to increase its accuracy, thus decreasing mispredictions and reducing the run-time of application. This reduction in run time is then sufficient to reduce overall processor energy by a larger amount than the increase in branch predictor energy.

Banking is introduced to the BHT to reduce the active portion of predictor and thus reduce dynamic power requirements. Banking increases the die-space requirements and hardware complexity but these considerations are stated to be beyond the scope of the paper. While the paper doesn't consider banking for the BTB it presumes that savings can be made.

The Prediction Probe Detector (PPD), introduced in [49], is used to gate access to BTB on non-branch instructions. The PPD is a separate table with the same number

of entries as the I-Cache. Each entry has one bit to control access to the direction predictor and one bit to control BTB access. The PPD needs to be accessed every cycle and must be accessed before the BPU. For the PPD to save energy it must prevent enough accesses to the BPU to outweigh the energy it consumes. As a result the PPD is more effective in a processor with a low associativity I-cache (when it becomes easier to detect if a branch instruction is imminent) and infrequent branches (as this results in fewer shifts between high and low power states in the BHT). It is also important for the PPD to be conservative in gating access to the BPU, as inaccuracies will lead to an increase in branch mispredictions which is highly costly. The results show that the PPD saves on average 30% of BPU energy and 3% of total processor energy.

Finally, the use of pipeline gating is considered and found to make little to no energy savings, even when gating is based on a confidence measure of the accuracy of in-flight branches. Errors in confidence prediction can lead to pipeline gating for correctly predicted branches, resulting in unnecessary stalling of the pipeline and increased program run time. Furthermore, pipeline gating is shown to save little in the case of a branch misprediction. These results only apply to the paper's chosen 'both strong' branch confidence technique.

Comment There is no mention of how the energy savings of PPD scale with the size of the PPD vs branch frequency. As mentioned, it is necessary for the PPD to gate enough accesses to the BPU in order to reduce the overall energy budget, however no real analysis of when this is achieved is made.

While spending more on the BPU to save CPU power is good for desktop processors, there may be times that this approach is not suitable for embedded processors. This is because of the extra hardware complexity and die-space required for the more complex, power hungry BPU configurations necessary to achieve ever increasing BPU accuracy.

While modern BPUs can easily have much higher energy requirements, as high as 50% of total CPU energy in some cases, it is important to consider: how far does this scale? It has been observed that branch predictors suffer from diminishing returns, where simply increasing the resources of a high performance predictor is insufficient to achieve any meaningful increase in performance and while creating hybrid predictors may give a further increase in performance that too will eventually reach a limit.

5. Power Efficient Branch Prediction through Early Identification of Branch Addresses, 2006 The approach of setting portions of a cache into a low power "drowsy" state has been shown to have applications in I-caches and D-caches, as well as in previous branch prediction papers. In [76] a new approach is presented where the distance to the next branch is obtained from static profiling information and used to wake up the branch predictor just in time to make predictions. This also has the added benefit of saving dynamic energy by eliminating BTB lookups in much the same manner as a PPD. Similar approaches to reducing BTB lookups are taken in [4] [5] [12] [13] [27] [36] [39] [48] [49] [50] [54].

The target program is statically analysed to allow for the next branch address to be calculated by finding the distance between the start of a basic block and the branch to be predicted. The Branch Identification Unit (BIU) is introduced to help calculate the branch addresses. This can be used to determine if an upcoming address is a branch instruction or not, and thus if access to the BTB is required.

Comment The requirement that the compiler must insert instructions to load the BIU before hot spots means that there will be some overhead in terms of extra instructions and so potentially an extended program run time. The profiling carried out by the compiler must be good enough to ensure that the hot spots must be sufficiently large or executed sufficiently frequently that the data in the BIU will not thrash back and forth. There is also a small complexity overhead from the requirement for a dedicated adder for next address calculation.

It is noted that GShare is designed to hash inputs in such a way as to use all BHT entries evenly. This may reduce the usefulness of a low power hibernation mode. This comes from the fact that it requires a large amount of energy to wake an entry from a low power mode. The result of a hash function that evenly distributes access across all entries would be that the average time an entry can be in a low power state would be reduced and the number of times it must be woken from a low power state would be increased, thus dramatically diminishing possible energy savings. As a result it would be worth considering whether it is better to use smaller tables where all entries are utilised and a drowsy technique is not needed (although this would probably lead to a rise in aliasing), or to construct a new hibernation-friendly hash function.

This approach essentially achieves the same as [50] by replacing the PPD with the new BIU structure. However, one advantage of this approach is that the next branch address must be calculated ahead of time. This potentially means that pre-fetching

of some manner, either for the I-cache or possibly the BPU, takes advantage of this information to help improve hit rates, thus reducing program run time.

The introduction of the drowsy hibernation mode is important improvement! As previously noted in chapter 1 there is a trend towards leakage energy being the dominant energy component in future processors. As a result, any saving to leakage energy could become a significant saving to the energy budget of the processor as a whole. Alternatively, such low power techniques could potentially be an important component in dark silicon considerations.

3.2.3 Compiler Hints

There is a class of predictors that aim to work smarter and not harder through the use of profiled hints obtained from the compiler. These hints are used at runtime to achieve a more accurate or energy efficient predictor.

6. Power-aware branch prediction techniques: a compiler-hints based approach for VLIW processors, 2004 Several of the papers already considered have used compiler hints to filter access to the branch predictor. In [46] a similar effect is achieved by detecting if the instruction bundle will include any branch instructions and if not prevents any access to the branch predictor tables. This reduces BPU access by up to 93%, giving 9% average energy reduction across the processor.

The BPU is held in a low power state until it is awoken ready for a branch instruction. This is achieved through the use of an optimising compiler. The compiler inserts a hint instruction ahead of time to enable the processor control logic to activate the BPU in time to make the prediction. The hint instruction contains the branch target (if the instruction uses an immediate address), a direction bit (if static prediction is to be used) and the cycles until the branch that requires prediction. This requires the addition of a specialised branch hint instruction to the ISA.

This technique is specialised in its application to VLIW processors by replacing NOPs in instruction bundles with hint instructions. If there are no NOPs to be replaced with hint instructions then no hint is inserted, in which case the branch falls back to a default prediction of not taken. This means that there will be no execution time overhead from inserting the hint instructions.

Comment A modern VLIW compiler can be highly successful at packing instructions together. This may result in there not being enough then NOPs in the instruction

stream to replace with the desired branch hints. The fall back state of predicting not taken will almost certainly result in less accurate predictions than may otherwise have been made by the dynamic branch predictor. As a result, the conflicting aims of removing NOPs and replacing them with hints may result in an increase in application run time due to the reduction branch prediction accuracy.

It would be interesting to pursue research on whether it would be possible to use several hints in a single instruction packet and if so whether it may even be desirable to devote an instruction packet to issuing multiple branch hints if there are no NOPs available for replacement. It would also be interesting to investigate the effect of altering the ISA to add hint bits to branch instructions. These would be used to aid prediction of biased un-hinted branches, although this would make the approach less flexible due to the changes in the ISA. It would also be interesting to see further research on the impact of altering the issue width to make more NOPs available for translation into hints.

Perhaps the most important message from this paper is that while this approach is clearly limited to VLIW processors it shows that it is possible to capitalise on specialised architectures to make savings that would not otherwise be possible.

7. Towards an Energy Efficient Branch Prediction Scheme Using Profiling, Adaptive Bias Measurement and Delay Region Scheduling, 2007 Static branch hints in various forms have been a proven technique for a long time, notably in [3] as well as in [10] [18] [21] [79].

In [29] a new approach is introduced, where adaptive bias measurement is used to dynamically assign static predictions for each branch. This is shown to reduce accesses and updates to the BPU by up to 62% and results in a global processor power saving of 6.22% on average.

The approach works by identifying a branch for static prediction if profiling shows a static prediction is more accurate than the dynamic predictor. This removes difficult to predict branches from the dynamic branch predictor and avoids the dynamic predictor's high misprediction cost.

While many ISAs allow for one branch hint bit this approach requires two. The two bits are used to encode one of four values which will control whether the dynamic or static predictor is used to make the prediction and whether the dynamic predictor needs to be accessed. The branch can either be marked as statically predicted taken/not-taken if it is sufficiently biased in the given direction. Alternatively, for unconditional

branches with an absolute address the delay slot is used to mask the latency of preparing the branch target and the dynamic predictor is not accessed. Finally, if neither of these conditions holds then the dynamic predictor must be accessed.

Comment The requirement for two hint bits allows little compatibility with old applications due to the requirements of a specialised ISA and may limit the uptake of the technique for new applications going forward. The technique also proposes the use of two delay slots, which is not often available. This is because it is generally difficult to find two suitable instructions to insert into the delay slots and because of the extra design complexity that comes with adding delay slots. Furthermore, delay slots are already commonly used to reduce the cost of all branch (mis)prediction penalties. The only contribution here is the gating of access to the dynamic predictor on branches where there is nothing to predict.

The presentation of possible savings to power is novel, but the choice made by the compiler during its profiling run may not match up with the goal of reducing power. During this profiling stage the compiler is seeking which is the more accurate predictor, there is nothing to show that this choice is also the best choice for reducing power or energy requirements.

8. Energy-Efficient Branch Prediction with Compiler-Guided History Stack, 2012
The design principle behind more modern predictors such as OGEHL and TAGE is the desire to be able to capture longer and longer branch histories to produce more accurate predictions (see chapter 2). The approach taken in [67] is to add a new compiler-guided history stack (CHS) to track very long distance branch correlations. The approach relies on the compiler to identify loop structures and procedure calls, and then to insert instructions around these to save and restore the global branch history using a stack structure.

The kind of very long distance correlations that the approach tries to capture are considered to be outside the capability of a dynamic predictor to capture at runtime. This is because running through something as simple as a single loop that iterates a sufficient number of times may easily be able to swamp the branch history register, filling it with information from that branch alone and thus dramatically reducing the ability of the predictor to make an accurate prediction.

The proposed response to this is to insert specialised save and restore calls around a loop or procedure call. This does not extend the branch history any further (a la

TAGE) but instead restores correlated information from other sources after the loop or procedure call is complete. The technique works on any history based predictor, including GShare, OGEHL and TAGE. The approach works best for limited history captured in small predictors, helping to remove aliasing by removing unrelated history.

Only 512 bits of storage and some simple control logic is required. The CHS implemented as a circular stack buffer, only accessed by these special load and store instructions.

Comment While the idea and implementation are elegant and simple, for the case of loops there are two possible issues. A simple loop predictor which gates accesses to the main predictor could do the same job, potentially more accurately (e.g. more accurate prediction of loop ending branches). Secondly, what if there is important information to be gained from branches in loops, especially embedded loops! For the technique to be successful the compiler algorithm must be good enough at producing code which can then be exploited by the technique.

This second point also holds for procedure calls. Even if the branch controlling data shares no dependencies it may be possible for histories upon returning from procedures to be distinct enough to signal different branch outcomes.

Better to do some profiling run and reapply instructions on case-by-case basis depending on what is found. Alternatively, what if direction predictors are not aliasing but procedure cleans out BTB, prediction is still inaccurate (basically trying to solve the wrong problem). This is however unlikely, as it has been shown that destructive aliasing is a large problem in many modern predictors.

3.2.4 Hybrid Predictors

9. Branch Classification: A New Mechanism for Improving Branch Predictor Performance, 1994 The use of branch hints to ensure that a branch is predicted in the most energy efficient or accurate manner is well researched. The approach in [11] is one of the earlier papers in this approach and yet goes a step further than most in associating a branch with the predictor best suited to it and ensuring the predictors are customised to ensure the best accuracy for the branches assigned to them.

Branches with the same static likelihood of being taken are split into the same class, with classes for strongly biased and mixed direction branches. Strongly biased branches benefit from short histories to allow for faster warm up of saturating counters,

while mixed direction branches benefit from longer branch histories to help distinguish different branch outcomes.

The paper notes that the use of static prediction for very strongly biased branches frees up the dynamic predictor to be specialised on moderately difficult to predict branches by extending the branch history.

To ensure that the predictions are made with the most suitable predictors, even in the presence of changes in bias behaviour as a result of different input data sets of changes of program phases, the decision of which sub predictor component to make the prediction is made by combining a static hint bit and dynamic 2-bit saturating meta counters.

Comment There is an important point made about not just targeting branches to the best predictors, but designing predictors to be best suited to their target branches. It is unfortunate that no investigation of power or energy effects are made in this paper.

10. Combining Static and Dynamic Branch Prediction to Reduce Destructive Aliasing, 2000 While previous papers considered in this chapter have considered the uses of branch prediction hints increase overall predictor accuracy and help reduce power and energy requirements, the approach in [53] goes further in its investigation into the relationship between the performance of the dynamic and static predictor components. Static profiling is used to help identify branches that should be predicted statically to help remove aliasing in the dynamic sub-predictor, claiming that for simple predictors using the technique has the same impact on performance as doubling the predictor size. Similar work is found in papers such as [19].

Offline profiling is used to identify branches that cause aliasing in the dynamic predictor, resulting in a drop in prediction accuracy compared to a predictor where this aliasing did not occur. Based on this profiling some branches are selected for static prediction, resulting in decreased aliasing and improved dynamic prediction performance. However, to get the best performance it is sometimes necessary to include the outcome of a statically predicted branch into the branch history register for the dynamic sub-predictor. This is because the outcome of the branch is useful in predicting other branches correctly.

The paper raises the usefulness of statically predicting both easy and hard to predict branches. This is because both classes of branches can cause aliasing in the dynamic predictor. Branches with a bias greater than a given cut-off value are statically pre-

dicted. Two hint bits are used, one to record the static bias of the branch and the other to select between the dynamic and static sub-predictors. Whether to shift the outcome of a statically predicted branch into the GHR can be selected on a per application basis or as a 3rd branch prediction hint. It is noted that branch biases can vary significantly based on input data set, as a result it is also possible to load different hint bits based on different input sets.

Comment The approach takes 2 or even 3 hint bits. This is fairly uncommon and thus not usually compatible with legacy code.

It is unfortunate that the energy and power implications of the approach are not considered. If well implemented the effect should be a reduction in dynamic power as a result of reduced accesses to the dynamic predictor and a reduction in static energy as a result of fewer mispredictions and thus a shorter running time.

The technique is shown to be effective for several compositions of similar hybrid predictors that work on essentially the same principles. However, the paper does not consider more up to date predictors such as Perceptron, OGEHL or TAGE predictors. All of these predictors still suffer from aliasing to some extent and as such may well also profit from the technique. In the cases where there is less aliasing to be avoided the predictors are generally more complex, resulting in more energy to be saved by avoiding dynamic predictions.

Finally, the idea of adapting the branch prediction hints based on the input set is not well explained, lacking details as to how the different data sets are detected and how a new data set is dealt with.

11. Low Power/Area Branch Prediction Using Complementary Branch Predictors, 2008 Many modern branch predictors aim to achieve the best possible accuracy with little consideration of other factors such as complexity, die-space or energy requirements. In [59] a case is made for the use of complementary branch predictors (CBP) as an alternative to such overly large, complex predictors. Instead, a small CBP is introduced to focus on frequently mispredicted branches, aiming to reduce the area required for a high performance BPU targeted at embedded processors. A 256 entry CBP improves processor energy efficiency by up to 23.6% and BPU efficiency by 97.8%.

The CBP works by tracking when the BPU will next mispredict and inverting the prediction. The PC of the last misprediction is folded and XORed with a concatenation of the GHR and the global misprediction history before finally being XORed with the

distance between last 2 mispredictions. Each entry stores 4 bits for PC (used as a tag check that the expected branch is encountered at the expected time), 8 bits to record the distance to the next misprediction, a single used bit (used for evicting entries) and a single bit for storing the correct prediction direction.

Although the technique sees some increase in prediction accuracy as a result of eliminating aliasing, such aliasing can be shown to be very low for larger GShare predictors where the technique can still reduce overall miss rates by around 50%. This is because the CBP is also good at detecting such common sources of misprediction as variable loop sizes and early loop termination.

An important result presented in the paper is that a static predictor used with a 1024 entry CBP is better than small GShare in some cases. The results demonstrate that for a 1KB bimodal or GShare predictor a 128 byte loop predictor is outperformed by a 128 byte CBP.

Comment The paper is concerned with aiming branch predictors at embedded processors and showing that it is possible and desirable to spend a small amount of valuable die-space on a dynamic predictor component. The selection of static, bimodal and GShare predictors with bits budgets of 0.25KB - 4KB give a good snapshot of the possible options for a range of embedded processor sizes. It is unfortunate that the paper dismissed OGEHL or TAGE predictors as too large or complex. It has been demonstrated that TAGE predictors can perform highly accurately at low bits budgets, such as is considered for the 4KB GShare with accompanying 1KB CBP, and may well be suited to the more powerful end of the embedded processor range. It would be interesting to see what effect adding a CBP would have on their accuracy and whether CBPs are as suited to all BPU types.

There are a few choices which make the results slightly harder to use. The first is that the paper allows 100M instructions to run to warm up the predictor to achieve more realistic results for how the predictor would act for the majority of the program. However, this avoids such important data as how the CBP acts during the warm up phase of the BPU, whether the use of the CBP has an impact on the length of the warm up phase or manages to reduce the high rate of mispredictions encountered during warm up. Furthermore, this decision may mask how well the CPU responds to changes in branch behaviours, possibly as a result of changes in program execution phases.

The energy figures are perhaps less dramatic than they appear at first glance. While most papers report the impact on process energy use, this paper reports the change in

energy efficiency - given as (misprediction rate) x (energy consumption per branch). Processor energy-efficiency is given as the energy delay product (ED).

One potential problem with the approach of using CMPs is that they will only ever prevent direction mispredictions. While these are some of the most common misprediction types, having a correct direction prediction without a correct target is generally not going to be sufficient. The BTB given for the results is a 512 entry 4-way cache, it would be interesting to see how results vary if this was altered.

3.3 Cooperation

12. A Study of Slipstream Processors, 2000 The Slipstream processor architecture [58] is based around the use of 2 cores to run the same application. One core runs a version of the application with a reduced number of instructions. This is achieved by identifying instructions that can be removed from the instruction stream without altering the correct execution of the program. This fast thread then passes back information to the second core which runs the original program to check that execution is correct. The information passed back allows the second core to pre-cache instructions and aid the BPU in its branch predictions, helping to speed up execution. As a result the cores run faster than a single core would.

This technique is only effective for highly limited bandwidth execution. This is because there is a risk that the selective instruction core runs too far ahead for the meta tables to capture enough information for the slower core. As a result the information passed to the slower core is unhelpful and results in slowing down the entire system.

Comment This technique requires the addition of large meta tables and very specialised hardware. As a result it is limited to working on pairs of cores and as a result is not generally flexible. There is also a very high overhead if a mistake is discovered in the fast running limited instructions core. As a result the technique is not suitable for highly parallelised programs where the chance of such mispredictions is too high compared to using the cores to simple run separate (perhaps speculative) threads.

The technique will likely result in increased energy and power requirements as the performance improvement may not outweigh the overhead of the meta tables and the requirement of having 2 cores running at once.

13. Accurate branch prediction for short threads, 2008 It has been shown that modern BPUs rely on long history lengths to achieve high branch prediction accuracy. Short threads such as those seen on speculative mulithreaded architectures destroy these long history lengths by filling the GHR with noise or unsuitable values. This is addressed in [14] by setting the GHR to the PC of the first instruction of the thread. This approach results in a reduction in mispredictions of 29% and an improvement in IPC of 13%.

During the warm-up phase where the saturating counters in the BHT are learning the correlation between history and branch outcome prediction accuracies are very low. During this phase it is only possible for the BHT to learn the correlations if the GHR contains meaningful values. In the case of speculative threads where it is unclear what the synthetic value for the GHR should be, or in the case of loading in new threads after a thread has been completed or offload to another core, the GHR is filled with meaningless values which must be replaced with branch outcomes from that thread before they become meaningful. While this is mainly a problem for long history based approaches such as GShare, OGEHL and TAGE, it can also affect Perceptron predictors.

The solution given is that when a new thread is loaded the GHR is initialised to the value of the first PC in this thread. This results in a reliably repeatable value for the case where the thread is run multiple times, giving the BHT a meaningful GHR value to work with.

Comment This is a highly successful technique that manages to recover inaccuracy imposed by speculation and is unlikely to cause any penalty to accuracy for longer running threads. It is important to note that the technique does not improve the best case (long running threads) accuracy, but that is not its aim.

While it would be possible to use special load/store instructions to deal with an interrupted GHR, it would be expensive in terms of performance overhead from the extra instructions and wouldn't address what GHR values to use for generated threads as nicely.

However, the technique is only really useful to multithreading where these short threads are likely to be encountered. Furthermore, the technique may be less useful for hybrid predictors that can avoid the prolonged warm-up period that this technique repairs. It is also noted in the paper that the effect of short threads on predictor accuracy was not an issue for all the benchmarks, some were affected by less than 3% when an

ideal predictor used. This raises the question of whether their benchmarks are generally easier to predict or are they less sensitive to this effect specifically?

14. The research of Multi-Core architectures's predictor, 2010 One of the few papers that currently addresses a multicore architecture, [34] targeting power and heat restrictions of global address predictors. This is achieved through the use of a shared pattern history table (SPHT) and a shared branch history shift register (SBHSR). These structures make it possible to reduce the size of the private predictor tables on each core, while improving accuracy and reducing power requirements.

This new approach is based on previous data reuse technology and aims to allow private pattern history tables to share information. This is achieved by the private predictors accessing a single large shared predictor (in many ways similar to a shared L2 cache). This allows for a longer pattern history to be used in accessing the large SPHT. To select the counter value to use the SPHT column is selected using j low-bits from the PC and k bits from SBHSR are used to select which row to use. This results in a global history, per-PC, 2-bit saturating counter being used to predict the branch outcome.

Each core's own BPU predicts the branch direction based on PC, fetching counter values for the branch if found in SPHT and updating SPHT on a misprediction.

Comment The main results graph is very unclear and at no point in the paper are any headline savings figures given. While the paper mentions that the private PHTs can be reduced in size, there is no indication of how large a reduction is possible and what the trade-offs are. Furthermore, there is no investigation into the energy requirements of the data that must be transmitted between the private predictors and SPHT, which could well be a non-trivial amount.

Finally, the paper only considers old predictor technologies such as GShare based predictors. No mention is made of how amiable the technique is to more modern predictors, such as Perceptrons, OGEHL or TAGE.

15. How to Implement Effective Prediction and Forwarding for Fusible Dynamic Multicore Architectures, 2013 The novel architecture proposed in [57] is based around fusing and splitting cores at run time to provide greater scalability as needed. Iterative Path Prediction is introduced to improve speculation accuracy. This is achieved through improved multi-exit block path prediction and exit-point prediction.

The EDGE architecture uses predicates to create large blocks of instructions where the outcome of a branch is less important than the location and timing of exiting to a new block. An OGEHL based predictor is used to perform next block prediction. Each core makes its own prediction, allowing fully a distributed workload at a budget of 8KBits across all cores. After each prediction is made the change in the GHR is broadcast to other cores to keep them all consistent.

Comment This technique is an example of where highly accurate BPU structures can be adapted to fit the needs of multicore operation through the use of a distributed prediction system.

3.4 The State Of Prediction Technologies

Taken together, the 5 papers in sections 3.2.1 and 3.2.2 present a wide variety of techniques that can be applied to reduce both energy and power requirements. Chapter 1 explained the importance of energy use both in the BPU and the processor as a whole. These papers have made large strides forward in addressing and reducing energy use. The challenge presented by such developments as multicore embedded chips (both in terms of their limited power supplies and strong thermal requirements) and the push towards smaller technology sizes, with the growth in the importance of leakage energy means that more work is required in this field to maintain the performance required.

The papers in section 3.2.3 have shown compiler hints to be a powerful tool in their ability to provide extra information to the dynamic predictors at runtime, allowing for increasingly accurate predictors to be built. The usefulness of compiler hints was further demonstrated in section 3.2.4, along with the power of using multiple different types of prediction mechanism to save power and get the most accuracy out of predictors by targeting them with the right kind of branches.

The range of papers presented in section 3.3 show the diversity of approaches taken towards addressing the challenges and opportunities that are presented by novel architectures. However, there have been very few papers that have attempted to tackle the novel challenge of multicore architectures. This is especially true for energy efficient approaches to multicore architecures.

This leads back to the central questions of this thesis, laid out in 1.9.1. By combining the kind of proven techniques shown in this chapter, chapters 5 and 6 of this thesis seek to show that better BPUs for embedded processors can be created. Chapter

6 goes one step further, with a detailed look at how heterogeneous predictors can be combined to give the best BPU configuration for the multicore embedded chip.

3.5 Summary

In this chapter the most up to date, accurate and efficient techniques available were presented. These have informed the selection of novel techniques that can be presented in this thesis and form a performance benchmark for the standard that such novel techniques must seek to improve upon.

The papers considered in this section often optimise for predictor accuracy or application run time. Chapter 4 shows that this often does not result in the best energy efficiency for the processor. Furthermore, by aiming to improve energy efficiency some further improvements in accuracy and run time can sometimes be obtained.

In the papers considering cooperation between cores there is little consideration of multiple cores running the same thread at the same time or within quick succession of each other. When such a scenario is considered it is with a complex and demanding solution that significantly adds to the die area and energy requirements of the system. Chapter 5 introduces the idea of a light weight system, requiring a minimal alteration to the existing hardware, capable of enabling multiple cores to cooperate in predicting branch outcomes for a shared application.

Chapter 4

Hybrid Static-Dynamic Prediction

4.1 Introduction

In chapter 1 it was shown that embedded processors must find the difficult balance of high performance in a cheap, small die-space, low power processor. In seeking to optimise all three of these it is generally discovered that optimising one generally comes at the cost of penalising the other two. The difficulties inherent in the use of global history branch predictors were presented in chapter 2, with GShare predictors identified as one of the most generally applicable techniques thanks to their relatively high accuracy and simple, scalable hardware implementation. A number of attempts to present a solution that delivers good performance with low energy and die-space costs were presented in chapter 3, with section 3.2.4 detailing the approach of several papers to the problem of efficient hybrid predictors.

This chapter presents a solution that reduces dynamic branch predictor aliasing, improving performance, reducing energy requirements and requiring a minimum of extra die space. This solution is achieved by taking a relatively well-known hybrid predictor, formed from GShare and static profiled BTFN sub-predictors, and investigating the decision of when each branch should be predicted either statically or dynamically. The approach differs from those given previously in its focus not on optimising performance at the cost of all else, but in attempting to give a range of solutions that prioritise energy efficiency while putting limits on the acceptable loss of performance.

The remainder of this chapter introduces the problems associated with hybrid branch predictors for embedded processors. The chapter then proposes a solution based on a novel parameter in the sub-predictor assignment process before presenting a motivating example, demonstrating this approach. A brief introduction is provided to the

ultra-low power target processor and the cycle accurate simulator used to collect our results before presenting the new approach to ultra-small dynamic branch predictors and introducing the use of a bias multiplier. The results of the novel biased predictor are summarised and analysed before discussing further work arising from these results. Finally the contributions of the biased predictor are summarised and linked back to the main contribution of this thesis.

4.2 Embedded Hybrid Branch Predictors

4.2.1 The Problem

Chapter 1 has shown that branch prediction can play a large part in the performance, die space and energy requirements of a processor. Without a branch predictor stalls are added to the pipeline, wasting time. While parts of the processor are not being actively used on a pipeline stall, components such as the instruction and data caches will still incur a large energy requirement due to the leakage energy inherent in their structures. As such, extra cycles result in extra energy consumption.

4.2.2 The Solution

Branch predictors are used to avoid these pipeline stalls, but come with their own hazards as explained in chapter 2. Generally the more resources given to the branch predictor tables the more accurate the branch predictor will be. This leads to a problem for very small predictors, such as are desirable for embedded processors, as the predictor tables would ideally be so small that their performance would be destroyed. As a result many embedded processors use static prediction schemes, such as BTFN. This is not a satisfactory result as BTFN usually achieves a low accuracy rate which this chapter seeks to show can be beaten by a carefully selected hybrid predictor, as illustrated in our motivating example in section 5.3

4.2.3 Past attempts

Previous attempts to solve the problem of dynamic predictors were presented in chapter 2. There have been attempts focused at low bits budgets featured in such places as the Branch Prediction Championship [33] where the Realistic Track involved a maximum bits budget of 32Kb+256 bits. This would comfortably translate into a GShare

predictor with a 2048 set, 2 way associative BTAC, an 8192 entry BHT, 14 bit GHR, and a 10 entry RAS. Using Cacti 5.3 [71] we can calculate this will take up somewhere in the region of 0.75mm^2 of die space at a 90nm implementation. The ARM Cortex-M3 (90LP implementation) [2] is a low-power processor designed for the embedded market. The Cortex-M3 takes up 0.12mm^2 die space and does not include a dynamic branch predictor, most likely for reasons of reducing die-space and energy consumption. If the proposed low bits budget GShare predictor were added to a Coretex-M3 it would result in a 625% increase in required die space, having a bits budget eight times that of the instruction cache or data cache. This serves to illustrate the large portion of die-space and energy that can be consumed even by a conservatively sized BPU.

In embedded applications this extra die space requirement means extra cost. If we assume the BTAC is accessed every cycle and add the dynamic read power to the leakage power of the BTAC and BHT we get a total additional power requirement of 23mW. When compared to the ARM Cortex-M3, which uses 16mW at 500MHz, this gives an increase of 143%. Such a large impact on energy consumption would drastically reduce the operating time of any battery operated device such a processor was used in. This is clearly not an acceptable solution and would never be considered for any processor design.

Most of the hybrid branch predictors presented in chapter 2 seek to optimise for performance, often at the expense of increased die area and/or energy requirements. A dynamic-static hybrid predictor can be different. It combines the use of a static predictor, such as BTFN, with the use of a dynamic predictor, such as the GShare predictor scheme. The dynamic predictor is only used when a branch is found to be too difficult for the static predictor to produce a reliably accurate prediction but not so difficult that the dynamic predictor often produces a costly inaccurate prediction.

Previous papers [29] [81] have shown that when a static-dynamic hybrid predictor is correctly used it results in performance better than that of a BTFN predictor alone. This is especially important when the dynamic predictor alone would perform worse than BTFN. This chapter extends this work with the novel contribution of an added bias multiplier which seeks to predict more branches statically, trading performance for energy efficiency. This approach also has the benefit of reducing the data that the dynamic predictor tables have to store, allowing for a reduction in predictor table sizes. This leads to a further reduction in dynamic and leakage energy consumption.

4.2.4 Novel Contributions

This chapter contributes a novel design space exploration of branch predictor bits budgets expressly targeted at embedded processors. The branch predictors produced are evaluated on the basis of die space, energy and performance. This chapter then goes on to propose a new and innovative trade-off between performance and energy savings, achieved through the use of a novel bias multiplier parameter which can be used to influence the number of branches predicted with the dynamic sub-predictor.

4.3 Motivating Example

Let us consider the following sample code in figure 4.1. In this simplified example the two branches loop_start and loop_end alias in the BHT, as shown in figure 4.2. In a two level branch predictor that uses a 2-bit saturating counter to predict branch outcomes destructive aliasing occurs when two branches with different outcomes are mapped to the same counter. This results in the counter being alternately incremented and decremented, giving inaccurate results.

A GShare predictor is a two-level branch predictor that uses an XOR of the GHR and the branch PC to calculate the BHT index. The predicted branch target is stored in the BTAC. If this specific code were run in a real predictor it is unlikely that aliasing would occur, but it is entirely possible to create pathological programs or poor indexing functions where very common branches like these will alias. We chose to illustrate this effect here through our simplified example.

```
             lw r1 3
loop_start   beq r1 0 next_block
             sub r1 r1 1
loop_end     jmp loop_start
next_block   lw r1 3
loop_return  jmp loop_start
```

Figure 4.1: An example of a simple loop structure in assembly

A simple BTFN predictor will correctly predict loop_end and loop_return all of the time, mispredicting loop_start 1/4 of the time. A hybrid predictor predicting loop_end and loop_return statically and predicting loop_start dynamically, given a sufficient his-

tory length and no other aliasing, will correctly predict all of the time after the warm up period.

To put this into real terms, we run the code sample from figure 4.1, with the loop_return instruction executed 120 times during program execution on a processor with a 5 stage pipeline and the previously-discussed GShare, BTFN and hybrid predictors applied. We assume there are no delay slots available, that the dynamic branch misprediction penalty is 5 and that the static misprediction penalty is 3. The saturating 2-bit counters used in the GShare components are initialised at 'weakly taken'. The global history length is 4 and that on a cold miss the GShare predictor falls back to a not-taken prediction.

When the GShare predictor is used we have a small number of mispredictions due to cold misses and warm up misses, but these will be greatly outweighed by the mispredictions owing to aliasing between the loop_start and loop_end branches. This is shown on the left of figure 4.2 at point (1).

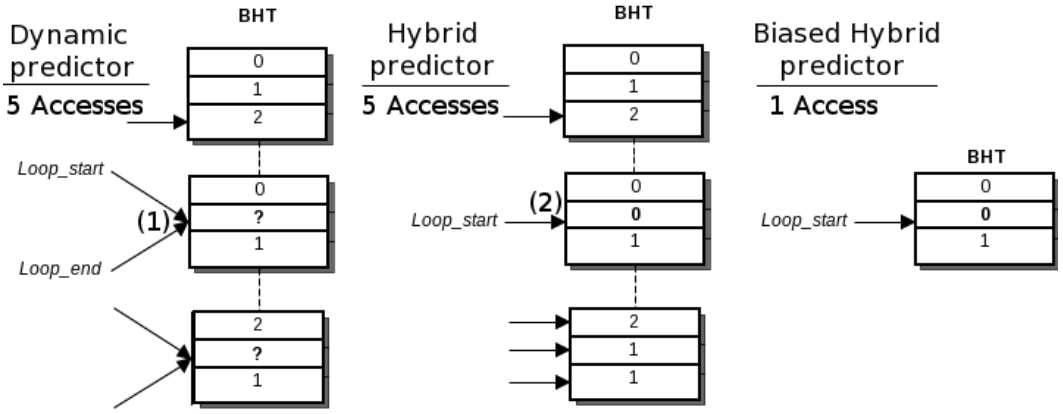

Figure 4.2: The benefits of removing branches from the dynamic predictor. Each box represents an entry in the BHT. Each arrow represents an access to a BHT entry. In the dynamic predictor two different branches access the same entry (1), resulting in destructive aliasing. In the hybrid predictor fewer accesses are made and the destructive aliasing is avoided (2). In the hybrid predictor where the bias multiplier was used further accesses have been removed and fewer BHT entries are required.

Considering just these mispredictions, there will be 720 mispredictions resulting in an additional 3600 cycles, or an increase in execution cycles of 142.8%. Larger predictor tables should help to remove this aliasing, but the aforementioned pathological programs and poor indexing functions can still cause aliasing even in a large predictor, wasting these greater resources, along with the associated die space and energy costs.

The BTFN predictor will mispredict loop_start 30 times, resulting in 90 extra cycles or an increase in execution cycles of 3.57%. The hybrid predictor will avoid the alias misses, as shown in the centre of figure 4.2 at point (2), only suffering four warm-up misses (the not-taken cold miss prediction is correct), for a total increase of 20 cycles, or an increase in execution cycles of only 0.79%.

The benefit of our introduction of the bias multiplier is shown on the right hand side of figure 4.2. Through the exploration of the design space using the bias multiplier it is discovered that we can reduce the size of the BHT (as well as the not pictured BTAC), thus reducing leakage energy and die space requirements.

4.4 Background

This section presents the architecture of the processor that was simulated when conducting our experiments. The experiments were conducted using an in house cycle accurate simulator. Some background on using a profiled BTFN/BNFT branch predictor is then presented along with the pitfalls of making a static/dynamic hybrid predictor.

4.4.1 The Processor

The processor simulated is based on the ARC EM6 [66], implementing the ARCompact™ ISA. The core features a 5-stage pipeline, both static and dynamic branch prediction, branch delay slots and predicated instructions. The simulated core makes use of a 32K 4-way set-associative instruction cache and a 32K 4-way set-associative data cache, both with a pseudo-random block replacement policy.

4.4.2 The Simulator

The simulator used to conduct the experiments is a cycle-accurate simulator verified against a register transfer level model, similar to that presented in [72]. This simulator implements the full-system: the processor, the memory system, interrupts and peripherals. The simulator was used to collect a number of profiling statistics necessary in evaluating the effectiveness of the technique.

4.4.3 Profiled BTFN And Hybrid Branch Prediction

The BTFN static prediction scheme, explained in chapter 2, has been shown to be relatively effective at predicting branch directions at a low cost [77]. However, for branches which are heavily biased in the wrong direction for this scheme it performs badly (e.g. a backwards branch which is rarely taken). In this case it is useful to employ a profiled BTFN/BNFT, where if the BTFN prediction for a given branch is correct less than 50% of the time then the prediction is inverted. This profiled BTFN predictor is used for the static component of our hybrid predictor and the baseline static predictor.

As shown in the motivating example in section 5.3 one of the benefits of using a static-dynamic hybrid is a reduction in the number of branches which need to be tracked by the dynamic predictor. One reason predictor tables need to be large is to avoid destructive aliasing, which leads to increased mispredictions (as explained in section 5.3). By removing branches from the predictor tables entirely, there are more resources for the remaining branches to utilise and prediction accuracies will rise as a result. Alternatively, the same predictor accuracies may be maintained and the size of the table reduced, giving important die space and energy savings.

Some architectures attempt to avoid aliasing by omitting unconditional branches from the BHT. In this simple architecture all branches are treated the same to avoid the computation of the address during the fetch or decode stages. It is important to note that we consider all types of branches and jumps (both conditional and unconditional) in our calculations as many unconditional branches can trivially be predicted statically.

One potential pitfall of this type of hybrid predictor highlighted in section 2.5 is the loss of information, specifically history information, available to the dynamic predictor. If the dynamic predictor relies on global history information to make its predictions (as do the GShare predictors used in this chapter), then executing a branch statically and not updating the GHR could result in mispredictions.

For example, a branch instruction with two regularly occurring outcomes could be accurately predicted using information in the GHR generated by a second branch instruction. If this second branch instruction is predicted statically and the GHR not updated, the first branch instruction accuracy will be affected, and in the worst case may drop to 0%. Clearly, when using such a hybrid predictor, it is important that secondary effects, such as updating the GHR or other structures such as the RAS, should be considered carefully to ensure maximum accuracy is obtained.

4.5 Methodology

This section explains the baseline figures used in the results comparisons, the workflow followed to produce the experimental results and the novel bias multiplier value introduced to the hybrid predictors. This chapter aims to build upon proven techniques, such as those presented in section 3.2.3. As such, it is assumed that the compiler can produce the required branch prediction hints. The ARCompact™ ISA used in simulations already features the sufficient scope for the required hint bits. The same dataset is used for both training and testing as such papers as [7] and [73] have shown that this is a safe approach for the general modelling required for design space exploration.

4.5.1 Workflow

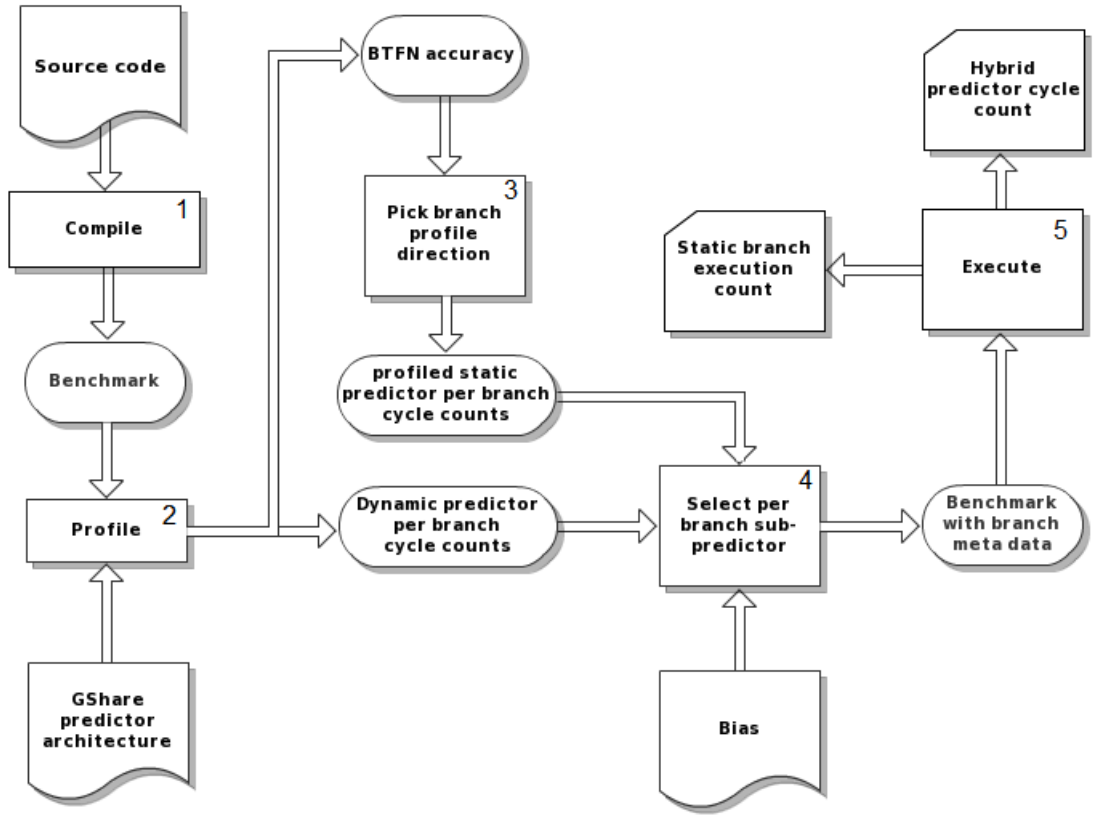

Figure 4.3: The work flow followed to conduct the experiments. Source code is compiled and the resulting benchmark profiled in our simulator. Each branch is given a profiled static predictor direction and then the selection between the static and dynamic sub-predictors is made. With this meta-data the benchmark is re-profiled and the resulting execution cycles and number of statically predicted branches collected.

A hardware verified simulator was used to produce the baseline execution cycle counts for running the EEMBC1.1 benchmark suite [70] on the target processor architecture using a static profiled BTFN predictor. The same figures for each of our hybrid predictor configurations were then collected, along with a count of the number of branch instructions (not dynamic branch instances) which were removed from the dynamic predictor and predicted statically. This work flow is shown in figure 4.3.

The static sub-predictor component of the hybrid predictor configurations consisted of a profiled BTFN/BNFT predictor which made use of profiling information from each benchmark. This online profiling was conducted by running the benchmark with the appropriate dynamic branch predictor architecture, predicting all branches dynamically (steps 1 and 2 in figure 4.3), and recording the outcome and direction of each branch instance. This information was then used offline to calculate the number of cycles that would be incurred by predicting the branch with a static BTFN predictor. If the BTFN predictor would have achieved less than 50% accuracy then a BNFT predictor was used instead, resulting in a static BTFN/BNFT predictor using profiling information on a per branch basis, tailored to each benchmark-architecture combination (step 3 in figure 4.3).

To discover whether a given branch should be predicted with the static profiled prediction or with a dynamic prediction the information collected in the previously-mentioned profiling runs was used to run an offline calculation of the number of cycles that would be incurred using the static and dynamic predictors. The inequality in equation 4.1 was designed to relate the total number of cycles resulting from using the static predictor to the number of cycles incurred by using the dynamic predictor (the bias value is explained in section 4.5.2). It should be noted that to make a static branch prediction requires key parts of the instruction to be fetched and decoded, including several delay cycles until the information becomes available and the prediction can be made. If the same prediction were to be done dynamically these prediction stall cycles are not incurred due to a combination of pre-decode bits and accessing the BPU every cycle. This is accounted for when computing the static prediction cycle cost.

$$static \leq dynamic + bias \tag{4.1}$$

In other words, for each branch instruction the cost of predicting the branch with static predictor (both misprediction cost and prediction stall cycles) is balanced against the cost of predicting the branch with the dynamic predictor (with its higher misprediction cost). The dynamic predictor is more costly in power and area than the static

predictor so an additional penalty is applied to the cost of the dynamic predictor predicting the branch. This additional cost is based on the bias multiplier and can be adjusted by the computer architect as desired. The branch is marked for static prediction is the results show that the static cost is lower than or equal to the dynamic cost.

If the inequality in equation 4.1 was satisfied then the branch was marked for static prediction, resulting in the branch PC and profiled direction being stored in a per-benchmark lookup table (step 4 in figure 4.3) that was then read into the benchmark the next time it was executed. At run time the current approach simulates obtaining this information from hint bits in the instruction set. This is the key step in the workflow, where the novel use of a bias multiplier is introduced. When the benchmark was re-executed (step 5 in figure 4.3) the branch instruction PC provided the key to this table and the prediction was made using either the dynamic or static sub-predictor component as necessary.

The dynamic sub-predictor component of the hybrid predictor configurations was created by selecting which GShare branch predictor dynamic predictor architecture to use (specified by the number of BTAC sets, the number of entries in each of these sets and the number of entries in the BHT) and the value of a bias multiplier parameter (further explained in section 4.5.2). The ideal configuration for the hybrid branch predictor would be one that is as small as possible, using as little energy as possible. It is desirable that the majority of branches are predicted statically, reducing active power consumption in the dynamic branch predictor. Finally, the overall predictor accuracy should be as high as possible, resulting in increased performance and decreased execution cycles.

4.5.2 The Bias Multiplier Parameter

This bias multiplier parameter is at the core of the novel contribution of this chapter and it is the use of this parameter which forms the core of the design space exploration. The bias multiplier was introduced to explore the boundary between performance and energy efficiency. The parameter takes the form of a very small multiplier, taking its value from a small percentage of the number of cycles the benchmark took when predicted using the dynamic predictor (collected in the previously-mentioned profiling run) and its value is calculated using equation 4.2.

$$cycles \times multiplier = bias \tag{4.2}$$

where *cycles* is the number of cycles taken to complete the given benchmark and *multiplier* is the bias multiplier parameter of the hybrid branch predictor configuration. This multiplier will be varied to trade accuracy for energy, moving branches from the accurate but power hungry dynamic predictor to the less accurate but lower energy static predictor. The multiplier is chosen to be some small percentage (thus having a range of 0 to 1) and the effect of varying it is shown in figures 4.6 - 4.9.

The bias multiplier parameter is used to bias the result of the inequality in equation 4.1 in favour of selecting a branch to be predicted statically. The key concept behind this is that the more branches which are predicted statically the greater the dynamic energy saving achieved through not accessing the dynamic predictor. Furthermore, when branches are predicted statically the dynamic predictor resources used are now freed up to predict the remaining dynamically predicted branches. This will in turn raise the accuracy of the dynamic predictor. If enough branches are removed from the dynamic predictor the predictor tables can be reduced in size without a large impact on performance, delivering a reduction in leakage energy and die space requirements.

4.6 Evaluation

The graphs in figure 4.4 and figure 4.5 provide a comparison of the arithmetic mean of the number of branches predicted statically and resulting execution cycles across the different combinations of dynamic predictor architecture and bias multiplier parameter. Here the 4 traditional dynamic branch predictor configurations (with a bias multiplier of 0) are compared against the 5th traditional predictor configuration, being the profiled static BTFN baseline. Each of the dynamic configurations is evaluated over 9 different bias multiplier values, giving a total of 36 novel configurations.

At the larger bias multiplier values nearly all of the branches are predicted statically, increasing from around 90% static execution to very close to 100%. The difference between the best and worst execution times is no more than 1.8%, with only a negligible difference in cycle improvements between the ultra-small 32 way BTAC and the 512 entry BTAC. This is possibly due to the large number of branches that are being predicted statically, meaning that the branches which remain to be predicted dynamically need very few resources to be so, accurately.

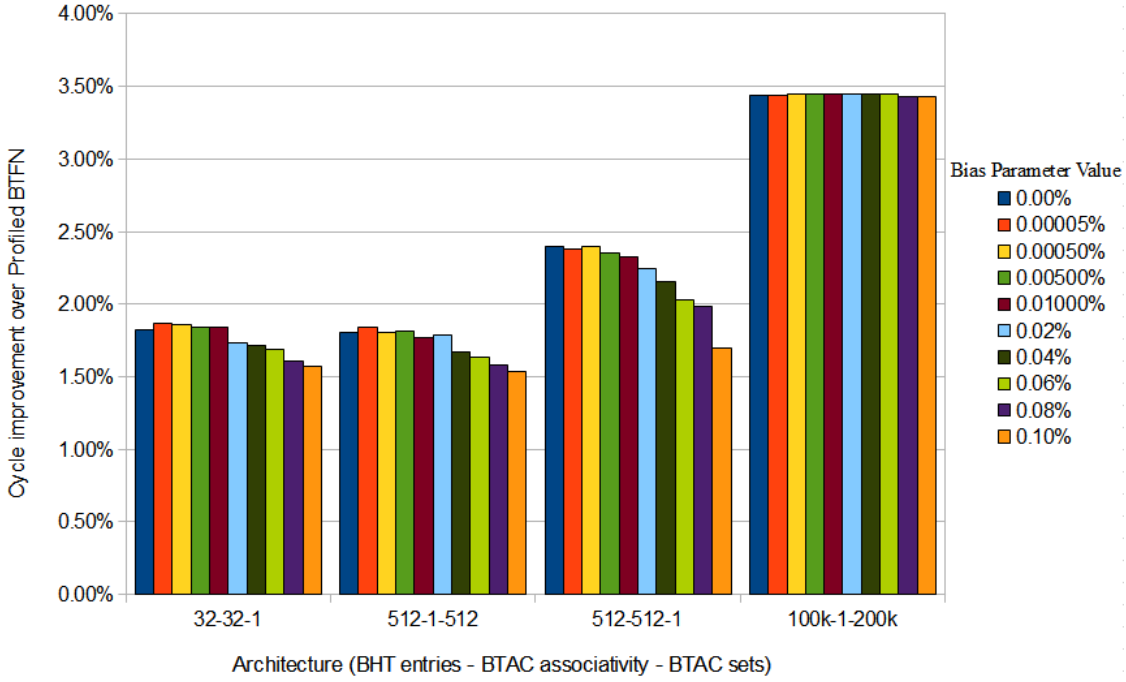

Figure 4.4: Execution cycles relative to a profiled static BTFN baseline. Higher bars show a greater improvement.

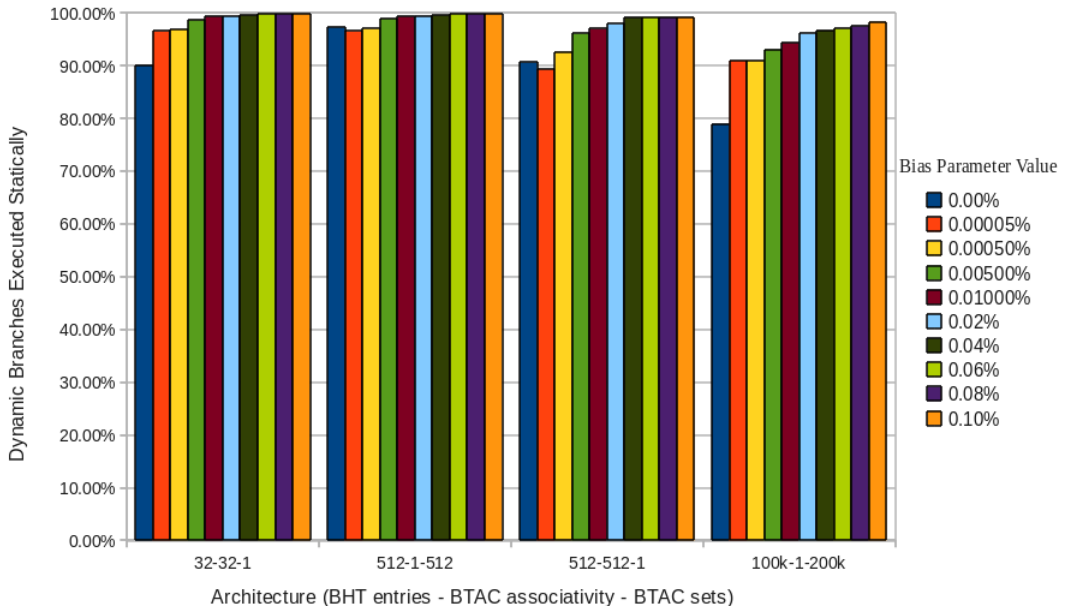

Figure 4.5: Number of branch predictions removed from the dynamic predictor and made with the static predictor instead. Higher bars show greater dynamic energy savings. Note how the change in bias multiplier affects the range of different predictor sizes, with larger predictors being especially sensitive.

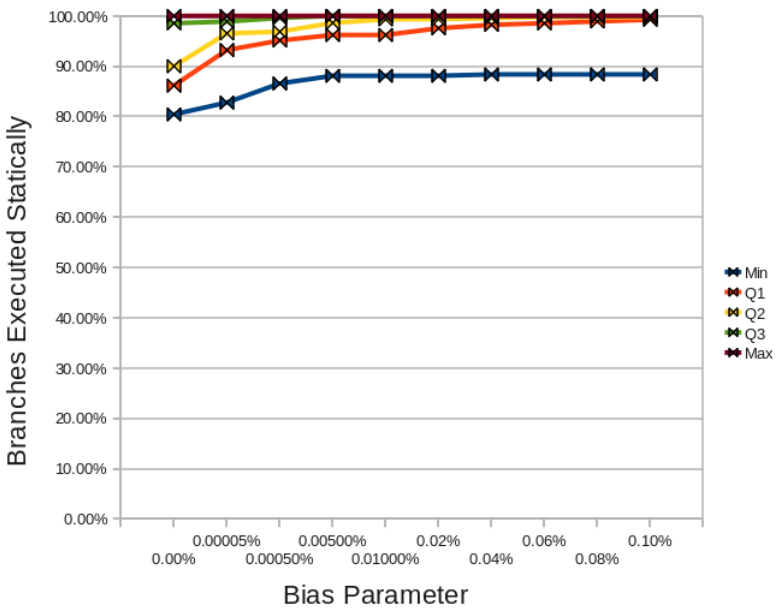

Figure 4.6: Spread of how many branches were predicted statically for a high performance dynamic predictor with 100,000 entries in the BHT and a direct mapped 200,000 set BTAC. Note how above a bias multiplier value of 0.01% Q1 to Max are close, but the minimum static executions series is significantly lower.

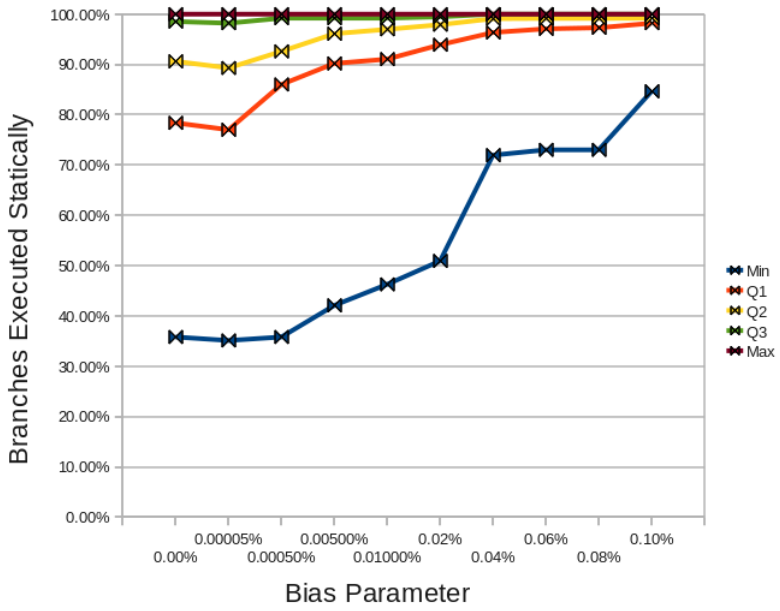

Figure 4.7: Spread of how many branches were predicted statically for a medium performance dynamic predictor with 512 entries in the BHT and a direct mapped 512 set BTAC. Note how the minimum execution series shows at least one benchmark performing especially well in the dynamic predictor, resulting in low static predictions, however there are steep rises at bias multiplier values of 0.04% and 0.1%.

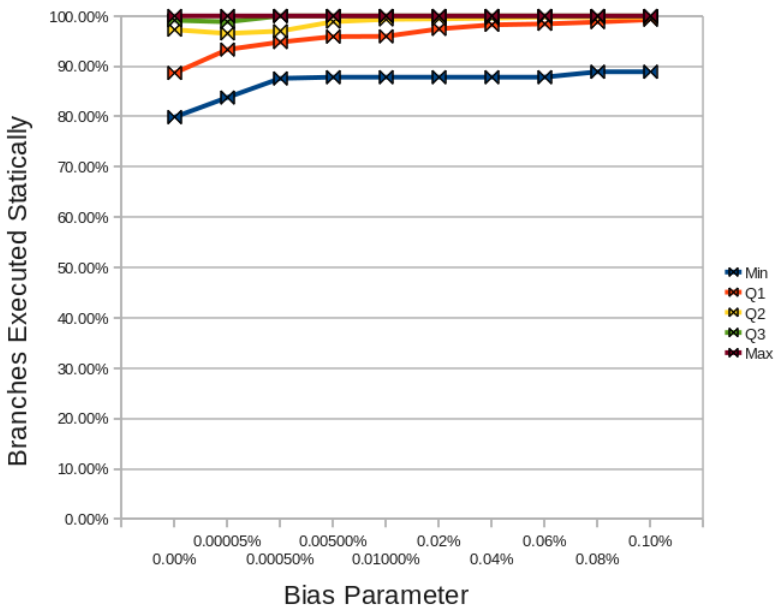

Figure 4.8: Spread of how many branches were predicted statically for a highly associative dynamic predictor with 512 entries in the BHT and a fully associative 512 set BTAC. Note the higher statically predicted branches in the Q2 series.

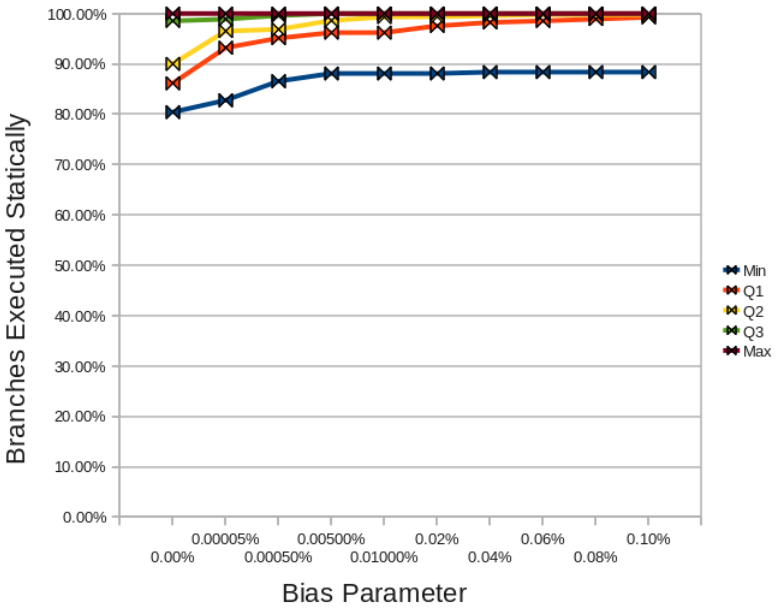

Figure 4.9: Spread of how many branches were predicted statically for an ultra small dynamic predictor with 32-entry BHT and a fully associative 32-set BTAC. Note the Min series stays just under 90% statically predicted for bias multiplier values above 0.005%, suggesting that even a very small dynamic predictor can outperform a static predictor for some branches.

The graphs in Figs. 4.6 - 4.9 show in detail the effects of the bias multiplier on the number of branch instructions predicted statically across the benchmark suite for each dynamic predictor configuration. Each graph shows a different traditional dynamic predictor configuration, with the associated BTAC size and associativity presented along with the BHT size in the captions, with the x-axis showing the exploration of the effect of the bias multiplier value. The lines on each graph represent the spread of how statically predictable the benchmarks were found to be, shown as the maximum, minimum, 1st, 2nd and 3rd quartiles. A small spread would indicate that all the benchmarks showed the same benefit for static prediction, while a larger spread would indicate that static prediction is more beneficial for some benchmarks than others.

The results show that the smaller the dynamic predictor, the higher the number of statically predicted branches and the smaller the improvement over the baseline. The novel result is that all of the different branch predictor architectures are shown to be sensitive to the value of the bias multiplier applied. The trend across the different architectures is that as the bias multiplier is increased the number of branches predicted statically increases, and the improvement over the profiled BTFN baseline decreases. To find the ideal bias multiplier, we look for the value which gives as close as possible to the top of figure4.4 and the bottom of figure 4.5. The best configurations are found with a bias multiplier value of around 0.02%, although this is higher for larger predictors and lower for smaller predictors.

4.7 Further Work

It has been previously noted in this chapter, as well as a number of papers in chapter 3 that the interaction between the dynamic predictor active power, dynamic predictor leakage power and the power consumed by the rest of the processor is a complex one. In order to get a detailed picture of the performance vs energy trade-off that this chapter started to investigate accurate energy figures are required. To achieve this a register transfer level hardware description language energy simulation model was sought, but proved to be too complex and time consuming.

The issue is further complicated by the fact that a single simulation cannot be used to sensibly model all possible predictor configurations. For example, for some of the ultra small predictor tables considered it would be better to model the entries as being stored in registers rather than a cache, especially when considering the control circuits around the cache requiring more energy than the cache itself.

4.8 Summary

This chapter has demonstrated that through building on well known branch prediction techniques it is possible to create a hybrid predictor with very low power and area requirements, suitable for modern embedded processors. This is achieved through a solution based on a novel parameter in the sub-predictor assignment process which explores the complex relationship between the use of the static and dynamic sub predictors. The key concept is that potentially sacrificing some branch prediction accuracy by accessing the dynamic predictor only when its resources are truly needed it is possible to reduce the pressure on the dynamic predictor, allowing it to be reduced in size.

While this approach does make strong requirements on ISA supplying sufficient branch prediction hint bits, this is an approach that has been used for some time. As a result there are popular ISAs targeted at embedded processors that already offer the desired functionality, allowing this new approach to be applied for a minimal cost in such cases.

In chapter 5, the focus shifts away from optimising a single BPU towards finding novel approaches to optimise the BPU for the multicore processors which are becoming increasingly popular.

Chapter 5

Peloton Branch Prediction

5.1 Introduction

In chapter 4 it was shown that branch predictors working in unison to form a hybrid predictor can often perform better than a single predictor alone. In chapter 3 it was noted that this idea of cooperation between different predictor structures has been key to the success of many different predictor types, not just hybrid predictors but also the more powerful single predictors such as L-TAGE which make use of multiple predictor tables working together. This idea of cooperation between prediction structures is central to this chapter.

The discussion of the state of modern processors in chapter 1 showed that the future of processors lies in multicore System on Chip (SoC) designs. These designs offer new opportunities for cooperation between processor cores. This chapter investigates the possibility of cooperation between the private BPUs attached to each core, resulting in each core working to aid the other and producing a more efficient system.

In revisiting the design of BPUs in the multicore era we propose a collaborative branch prediction scheme called *Peloton* for data-parallel workloads, which enables efficient communication of branch prediction information across cores. This technique is effective on simple techniques such as GShare, as well as more complex TAGE predictors.

In this chapter we demonstrate the power of communicating data between BPUs to reduce miss rates for data-parallel workloads. We propose a technique for communication between these BPUs, which can be implemented efficiently in hardware. We evaluate the effectiveness of our Peloton scheme using the data-parallel PARSEC2 [6] benchmarks and a detailed prototype implementation of our scheme in the MARSS

instruction set simulator [52]. We present a brief design space exploration demonstrating that Peloton branch prediction is applicable for a range of branch predictor configurations.

The remainder of this chapter presents a motivating example, demonstrating the benefits of our novel approach, followed by a brief study into the scope for improvement offered by Peloton branch prediction in an ideal environment. We then introduce the target processor and the cycle accurate simulator used to collect our results. Using this context we analyse the effectiveness of our approach to see where gains can be made and losses avoided. We then outline how Peloton branch prediction may be realised in hardware and present the best results our simulations produced. The effectiveness of Peloton branch prediction is further explored through an exploration of alternative architectural configurations and branch predictor types before discussing further work arising from our results. We finally conclude with a summary of Peloton branch prediction, its effectiveness and what work remains to be done.

5.2 Peloton Branch Prediction

Multicore processors are increasingly common in the latest computer architectures, even down to the level of small, energy efficient embedded devices. This move towards multicores being everywhere puts an increased emphasis on needing to get the best performance out of multiple cores operating together. While some performance gain can be realised by operating in parallel across several cores, single-core performance is still an important aspect worthy of optimisation.

One key aspect in straight line performance is the accuracy of the BPU. An inaccurate BPU leads to increased program cycles, meaning reduced throughput and greater energy consumption. While modern BPU designs can be highly accurate (in some cases in excess of 99% [4] [5]), it is difficult or undesirable to fit these designs into the low power, low area, low cost constraints of embedded multicores.

While many papers have focused on optimising branch prediction for the single-core case and confidently asserted that their approach will perform well in the multicore case, few papers have tried to directly address and optimise for the unique resources and opportunities provided by a multicore platform. This chapter presents a novel approach to branch prediction for data-parallel workloads on multicore platforms by broadcasting branch prediction information between BPUs.

Using a data-parallel programming paradigm in a framework such as OPENMP, the

programmer marks up data-parallel loops, where (ranges of) iterations are distributed over a number of threads. Each thread executes a different section of the loop, with all the threads executing in parallel on a multicore host. It is important to note that in such a data-parallel programming model all threads execute the *same* task, but operate on different data. Given the high correlation of control flow between cooperative threads in a data-parallel configuration, our hypothesis is that branch prediction information is also correlated and sharing of this information between BPUs has the potential to reduce branch mispredictions and, ultimately, improve performance. The questions we are trying to answer in this chapter are how a practical scheme for sharing branch predictor information on a multicore machine can be implemented and how this can be made efficient with respect to performance and energy consumption.

Throughout the chapter the Peloton technique will be compared to a baseline of running a system that is identical except for the use of updates being shared between BPUs (unless otherwise stated). A successful and efficient solution is one which results in a saving in application runtime with little to no increase in energy cost.

5.2.1 Slipstream Processors

The technique presented in this chapter is similar in concept to the idea of a slipstream processor [58] [65]. However, the implementation of the concept is very different. A slipstream processor makes use of one run-ahead thread to discover information on easy to predict branches and instructions with no impact, with the aim of sending this information to a second thread which can run faster as a result. This leads to a reduced execution time for the thread running the whole program. It is necessary for the thread running the reduced program to be running on a fully functional processor to slow down execution in the case of bad information being passed from the reduced thread.

As a result a slipstream processor requires two processor cores (or properly scheduled SMT processor) to speed up a single application. Furthermore, a slipstream processor requires a number of new caches and communications controllers to facilitate the information being passed between the two threads. As a result the overhead in terms of hardware required to run a single application is very high. In contrast, Peloton branch prediction does not tie up a processor core in producing predictions, allowing for a higher throughput (especially important in the type of data-parallel workloads that this chapter is concerned with). Furthermore, there is no requirement for additional caches to store information, only access to the existing interconnect.

5.2.2 What Is Peloton Branch Prediction?

We draw a parallel between our contribution and the approach seen in a flock of birds or a large group of cyclists (called a Peloton). Each individual spends an amount of time at the front of the group, doing the hard work while the others behind have an easier time. Similarly we can apply this thinking to the case of two or more cores running threads from the same data-parallel loop and have one or more cores that run behind the first. The BPUs of the cores running behind the first core can make an easier prediction (i.e. one which is more likely to be accurate) as a result of having information passed back from the first core. This will dramatically reduce capacity, conflict and warm-up misses, as well as reducing the effect of destructive aliasing in predictors based on global history.

For the approach taken in this chapter it is important that the cores have homogeneous BPU configurations such that prediction information can be passed as compact indexes into the history tables, rather than full-blown table entries. Using our sharing scheme, cores which are running behind will make fewer branch mispredictions, experiencing the slipstream effect, and thus may overtake the original first core. At this point it is important for the new first core to have an accurate branch predictor so that the information sent to the other cores is as useful as possible. In this way the average branch prediction miss rate of each of the cores and the overall branch prediction miss rate will be reduced.

5.3 Motivating Example

To illustrate the key idea of our work we now present a motivating example to demonstrate how Peloton branch prediction results in improved branch predictor accuracy. The example shows the progress of two cores through a simple application, with the y-axis representing time (i.e. further down the image is further through the application).

On the left we see the two cores are not communicating their branch predictor updates. As a result both cores reach the same branch with the same state in their branch predictors and consequently both make the same misprediction. On the right we see that once core 0 has updated its branch predictor the update is sent on to core 1. This allows core 1 to update its predictor with knowledge of the branch it is about to reach. As a result when core 1 does reach the branch it predicts correctly.

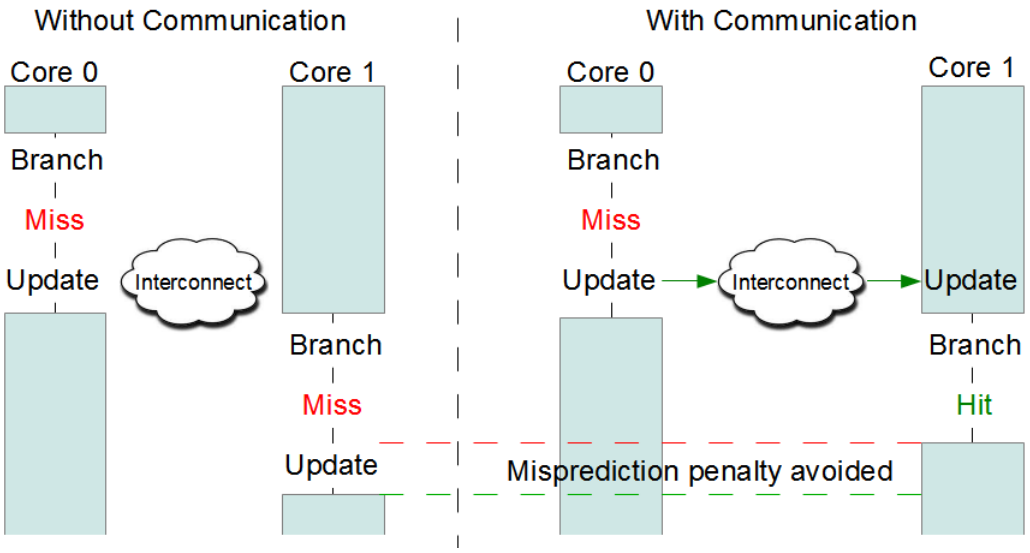

Figure 5.1: Demonstrating the way that sharing information between branch predictors can lead to fewer mispredictions.

Information is shared between the BPU units on different CPUs in a model broadly based on a cache coherency update. The occasions at which updates will be sent between the BPUs are dependant on the communications strategy, further explored in section 5.6.5. When it is decided that an update needs to be sent the BPU sending the update broadcasts the relevant information across a local bus, potentially the pre-existent data bus or a new dedicated bus. A more detailed description of the exact update mechanism and the data transmitted is presented in section 5.7.3.

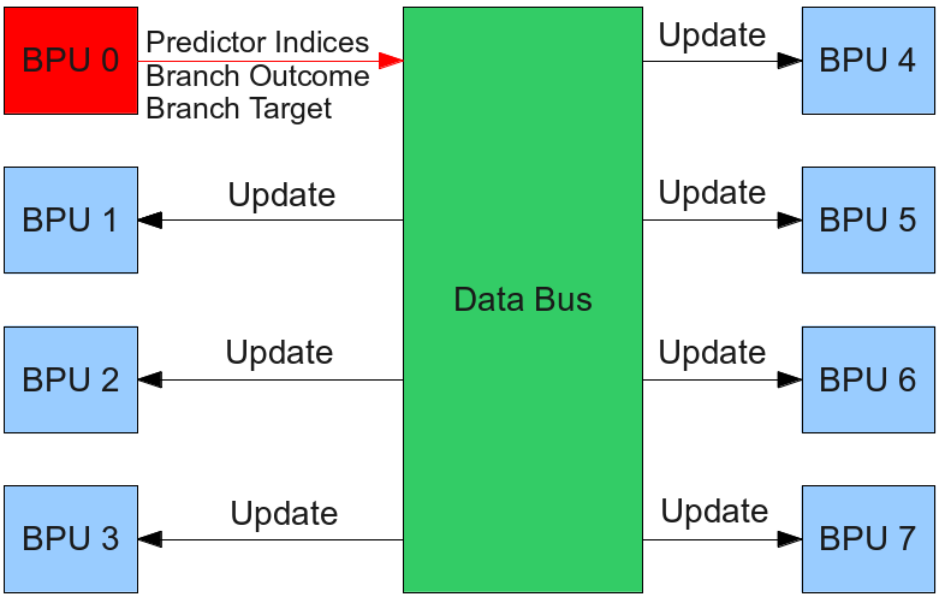

Figure 5.2: Each BPU uses the data bus to broadcast updates to all other BPUs.

5.4 Background

In this section we present the architecture of the processor that was simulated when conducting our experiments. The experiments were conducted using the MARSS simulator.

> "MARSS uses a cycle-accurate simulation models for out-of-order and in-order single core and multicore CPUs implementing the x86 ISA. These are integrated into the QEMU full system emulation environment." [52]

We chose to simulate the PARSEC2 benchmark suite [6], containing examples of demanding, data-parallel workloads, suitable for research into diverse, non-high performance computing applications. We used the simsmall dataset to simulate only the benchmarks described as data-parallel. We used the built-in Intel Atom processor model as a representative model of an in-order core tested against real hardware [43].

The modelled core is single threaded, with 2-wide fetch and issue width, 2 integer, 2 floating point and 2 complex functional units, 32-entry commit buffer and 16-entry dispatch queue and store buffer. Each core has a 256-set, 8-way MESI instruction cache and an identical data cache. A single 2^{12}-set, 8-way L2 cache is shared amongst the cores. The benchmarks were each set to run 4 threads over the 8 single threaded cores. The branch misprediction cost was 6 cycles.

The branch predictor modelled comprised of a 2^{10}-entry, 4-way BTB, used to predict branch targets and a 2^{10}-entry RAS used for storing call return addresses. The RAS was implemented as a circular stack, with each of the entries storing information about a call-return pair. Each time a call is made a new return address is pushed onto the top of the stack. The number of entries is chosen to be far larger than is likely to be needed to ensure that the performance of the RAS does not hinder the overall accuracy of the BPU, allowing the focus to remain on the performance of the BTAC and the BHT. Each of the RAS entries comprises the 31 bits required to capture the branch return address. The direction predictor was a hybrid GShare-Bimodal predictor with a bits budget of 64KB to facilitate comparisons with other papers.

Each of the hybrid's two subpredictors (GShare and Bimodal) had 2^{16}-entries, and were accompanied by a 2^{16}-entry meta predictor to predict if a branch is taken. The GShare predictor uses a global branch history register to store the outcome of all branches, which is then XORed with the PC of the branch to calculate the index of the 2-bit saturating counter used to predict the branch outcome. The Bimodal subpredictor uses no history information and indexes its 2-bit saturating counters using

the branch PC folded in half by XORing. The meta predictor was implemented in the same way as the Bimodal predictor.

5.5 Limit Study

As a limit study of what might be achievable the PARSEC2 benchmark suite [6] was simulated, assuming latency free transmission of information between BPUs. Figure 5.3 suggests that there is large scope for branch prediction accuracy improvement across the benchmark suite.

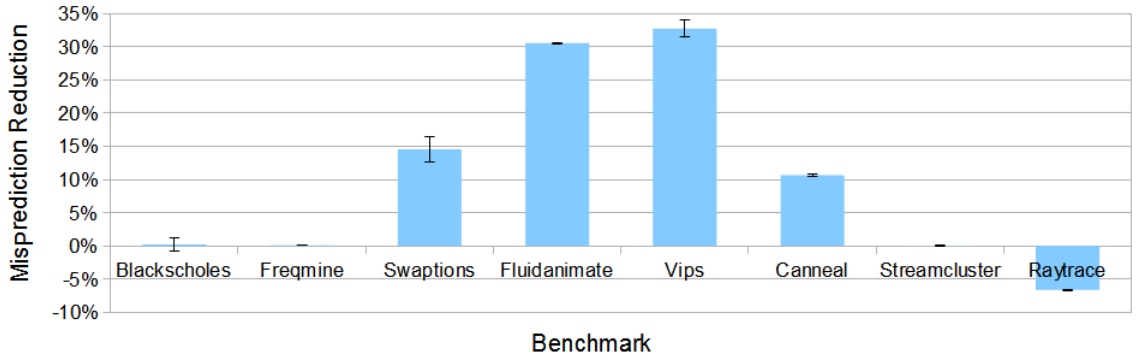

Figure 5.3: Demonstrating the effect of Peloton branch prediction on a range of data parallel benchmarks. Each benchmark was run 10 times and the average taken. The baseline is the same architecture but no updates are shared between BPUs. The error bars shown are the 95% confidence interval from the standard error.

Benchmark	Unshared Miss Rate	Shared Miss Rate
Blackscholes	5.70% (± 0.04)	5.69% (± 0.07)
Freqmine	7.13% (± 0.00)	7.12% (± 0.00)
Swaptions	5.40% (± 0.02)	4.62% (± 0.15)
Fluidanimate	12.97% (± 0.03)	9.01% (± 0.00)
Vips	1.23% (± 0.00)	0.82% (± 0.02)
Canneal	5.44% (± 0.01)	4.86% (± 0.01)
Streamcluster	1.02% (± 0.00)	1.02% (± 0.00)
Raytrace	1.16% (± 0.00)	1.23% (± 0.00)

Table 5.1: Miss rates for GShare limit study with and without update sharing

However, figure 5.3 also shows that several of the benchmarks may suffer from the approach as well. As such it is necessary to consider what it is that makes Pelo-

ton branch prediction beneficial, how branch mispredictions may be reduced, when updates should be sent and to which cores.

5.6 Branches in Data Parallel benchmarks

This section assesses a number of factors in the performance of Peloton branch prediction. First the frequency of sharing between BPUs is analysed. The impact of Peloton branch prediction on different branch types is then considered along with an analysis of what miss types are reduced by the technique. The presence of the expected slipstreaming behaviour is then analysed before combining the information presented in this section to present some new communications methods and revist the requency of updates between BPUs.

5.6.1 Write Frequency

To analyse the potential of how much of an impact Peloton branch prediction would have on the interconnect we collected data on how often updates were transmitted. To collect this data we ran MARSS for intervals of 100 cycles and counted the number of incoming updates. This was repeated throughout the entirety of each benchmark. These data were then split into classes based on the average number of updates in each interval. Finally, the data were converted into a cumulative frequency graph. This allows an assessment of how many updates the BPU must be able to handle per cycle against how frequently this would be sufficient.

In figure 5.4 a selection of benchmarks is used to present the different behaviours seen across our earlier presented set of simulated benchmarks. While Raytrace shows 90% coverage of the intervals with an average of 1 remote update per 10 cycles, to achieve the same coverage in Freqmine requires an average of 3 updates every 2 cycles.

This is a troubling result for two main reasons. Firstly, each update is taking up interconnect bandwidth. This potentially means that so many updates could be sent between BPUs that program execution is slowed due to the interconnect being swamped by these updates. Alternatively, the interconnect could be widened or a specialised BPU interconnect added. Both would add a level of hardware overhead that was unacceptable for the work considered here. Secondly, the cost complexity and die area required for having a multiple ported BPU are large and grow rapidly with the number of ports.

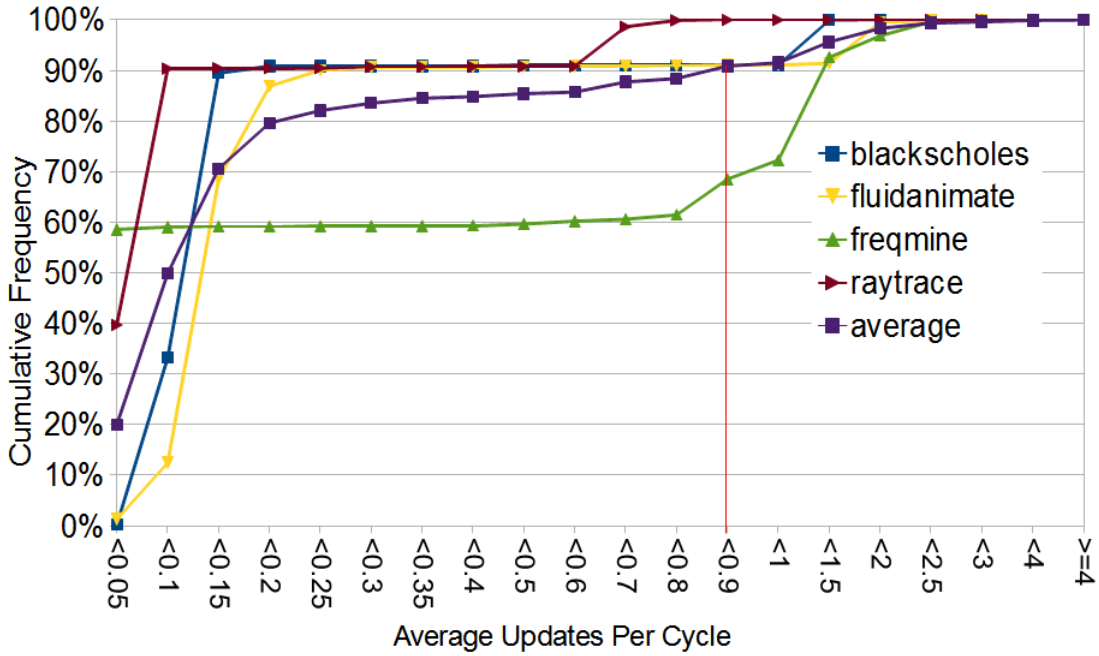

Figure 5.4: The number of remote updates arriving was counted over 100 cycle periods and then divided by 100 to give average updates per cycle. The cumulative frequency on the y-axis is calculated by counting all 100 cycle periods where the average updates per cycle is less than or equal to the value given on the x-axis, then dividing by the number of 100 cycle periods. Results are presented for a range of benchmarks and an average across these benchmarks.

In order to address these problems further work was carried out into when updates should be sent and which BPUs should receive them. This involved such considerations as only sending an update when the BPU has mispredicted or only sending an update when the BPU counters are not saturated (i.e. when the update will result in changing counter values). The full results of this work are presented in section 5.6.5, with these and similar approaches reducing the number of updates sent at a potential cost to misprediction reduction.

5.6.2 Branch Types

To further study if and why Peloton branch prediction is effective we now present a breakdown of how each branch type is affected by it. In order to do this the technique was applied to each branch type individually. The effect on miss ratio is shown in figure 5.5.

Unconditional branches are always taken, while conditional branches may be taken

depending on a condition in the branch instruction. Direct branches are branches with absolute target address, while indirect branches are branches with a target stored in a register, or with a variable offset, so that the target may change between executions.

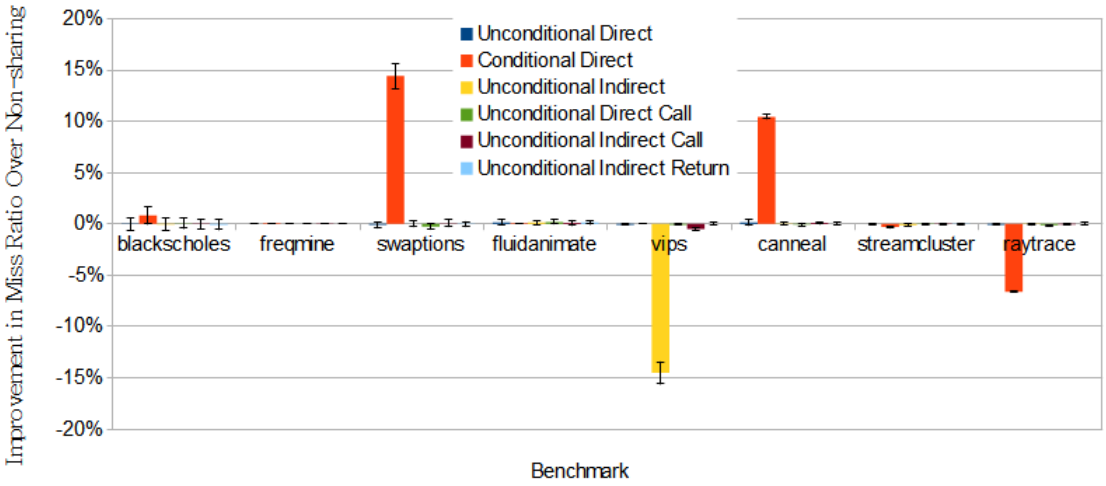

Figure 5.5: Impact on miss rate as a result of enabling the single specified type of branch, dependent on the results series. A negative number shows an increase in mispredictions. Each type of branch was shared individually to highlight the impact of Peloton prediction on that type of branch.

Looking at this breakdown of the impact of each branch type, we can see that the only types to have any impact (greater than a single standard deviation) are conditional direct branches and unconditional indirect branches. It makes sense that the branch call and branch return predictions show very little impact from the use of Peloton branch prediction as a combination of the instruction and a sufficiently large RAS will contain enough information to predict these instructions with a high degree of accuracy. There is very little information that could profitably be sent between the BPUs, and a high chance of sending incorrect information. Likewise, the unconditional direct branches are trivial to predict and so sharing prediction information between BPUs is a waste of energy.

The conditional direct branches are shown to be highly suitable for Peloton branch prediction in several of the benchmarks. This is likely due to these branches accounting for loop end branches and if statements. The outcome of these branches may well be data dependent or otherwise strongly correlated with progress through the parallel loop. This would suggest that the behaviour across different but similar data points being computed on different cores will be very strongly correlated and so ideal for Peloton branch prediction. Unconditional indirect branches (which are not calls or re-

turns) are more likely to be data dependent and as such there is a greater likelihood that the branch behaviour will vary from core to core as they execute different data points. Therefore, Peloton branch prediction would be unhelpful at best and may reduce performance in the worst case. This is seen in the large reduction in accuracy in the Vips benchmark. We take advantage of this result in section 5.6.5.

5.6.3 Classification Of Misses

It is important to understand how Peloton branch prediction makes an impact on branch prediction accuracy. To this end we collected data on the basic miss types of cold, direction and target misses. Cold misses are misses as a result of a branch that has never been seen by the predictor and as such there is no data to predict them on. Direction misses are where the branch has been encountered before and the outcome of the branch is mispredicted. These can only occur when a cold miss does not occur. A target miss occurs when a cold or direction miss have not occurred but the predicted target for the branch is incorrect.

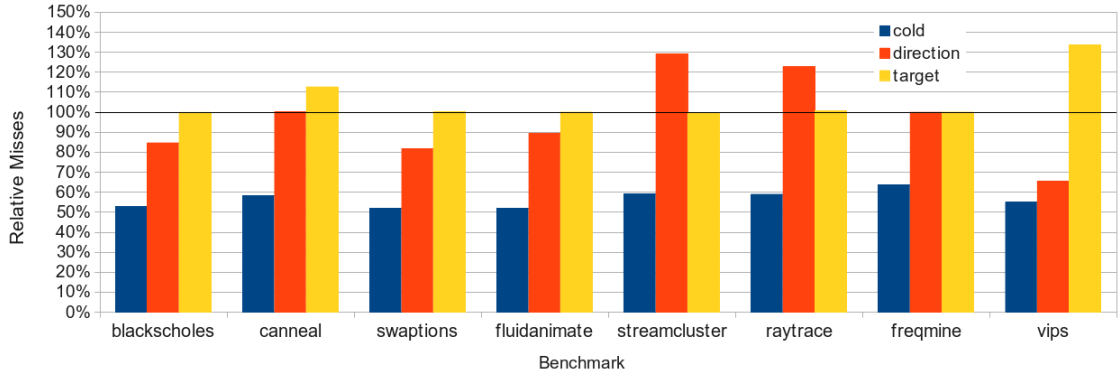

Figure 5.6: Miss types across the benchmarks using Peloton branch prediction compared to non-sharing. The baseline at 100% is the number of the given type of misprediction observed for the given benchmark when Peloton branch prediction is not used.

Figure 5.6 shows the number of misses of each type for each benchmark compared to a system that does not share information between BPUs. For each benchmark the number of cold misses is roughly halved. This is as a result of information being sent to cores that are yet to encounter the branch in question. This results in what would have otherwise been a cold miss being replaced with a correct prediction.

The number of direction misses varies between benchmarks, with a range of a 20% reduction to a 30% increase. An increase in direction misses is possibly due to three

different effects. The first is that the increase in updates results in more destructive aliasing between branches. The second effect is that the balance of conditional to unconditional branches is significantly different between the benchmarks where direction misses decrease and those where direction misses increase. A larger number of conditional branches will also increase pressure on the direction predictors. Lastly, conditional branches where the branch condition is data dependent may be poor candidates for Peloton branch prediction, with updates between BPUs moving the saturating counter in the opposite direction to that required by the local core.

The number of target misses is generally the same, with the exception of Canneal where the number of target misses increases. This could be due to the branch targets being unpredictable and highly data dependent, such that sharing information between BPUs is not helpful in reducing mispredictions. It is also possible that while some target misses are being removed, mispredictions that were previously direction misses are now mispredicting the target and becoming target misses.

5.6.4 Slipstreaming

We now turn our attention to an analysis of whether the cores behave as we expect with respect to the different rates of progress through the program. We tested whether different cores are displaying the slipstreaming behaviour of taking turns at being producers or consumers of updates. We collected the number of updates sent and updates received over 100,000 cycles and formed a ratio of these numbers by dividing updates sent by updates received.

A core which has made more progress than other cores will have not received branch information before it reaches a branch and is more likely to mispredict. Since cores only send updates after a misprediction a high number of updates sent will suggest a core that is ‘out in front’ of the others. Conversely a core with few updates sent out and many updates received suggests that the core is following the progress of a different core and benefiting from the updates received. By counting how often a core has the highest ratio amongst all eight cores per time slice we can suggest how often cores are overtaking over cores and changing their order.

Figures 5.7 - 5.12 show the amount of time that each core spends as the core with either the highest or lowest ratio. The higher the frequency the higher the number of 100,000 cycle periods that the core occupied the position as the core with either the highest or lowest ratio. The sum of the frequencies may not add up to 100% (and often

does not for the figures showing the lowest ratio). This is due to times when two or more cores share the same ratio.

Looking at Blackscholes in figure 5.7 we can see that core 1 consistently has the highest ratio. This suggests the core is progressing ahead of all the other cores. Conversely, we see in figure 5.8 that cores 5-8 are all roughly equal in their share of how often they are the core with the lowest ratio. Looking at Fluidanimate in figure 5.9 we can see cores 1, 2 and 4 all have a large share of time as the core with the highest ratio. In figure 5.10 we again see a very similar picture to that observed with Blackscholes. Looking at Vips in figure 5.11 we can see core 5 dominates the time spent with the highest ratio, although cores 1, 2 and 7 each spend more than 10% of the time with the highest ratio. In figure 5.12 we again see that cores 1, 6 and 8 each spend a significant time as the core with the lowest ratio. These results suggest that the cores often behave in the expected manner with several cores taking it in turns to be at the front.

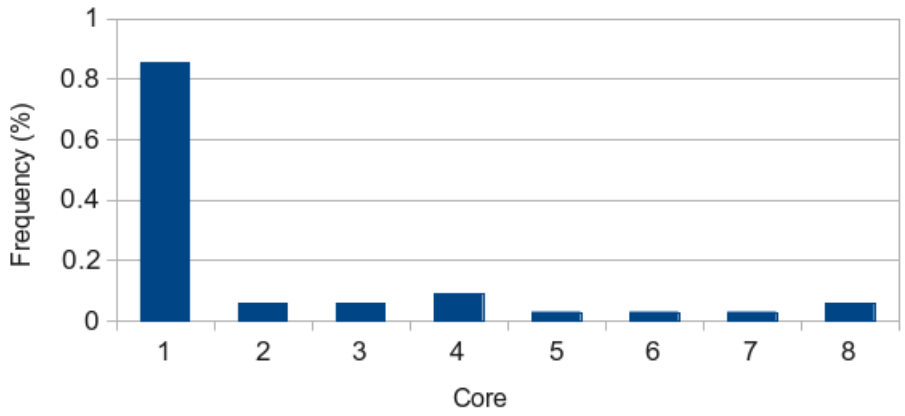

Figure 5.7: Frequency of time a given core spends with the highest ratio of updates sent to updates received for the Blackscholes benchmark

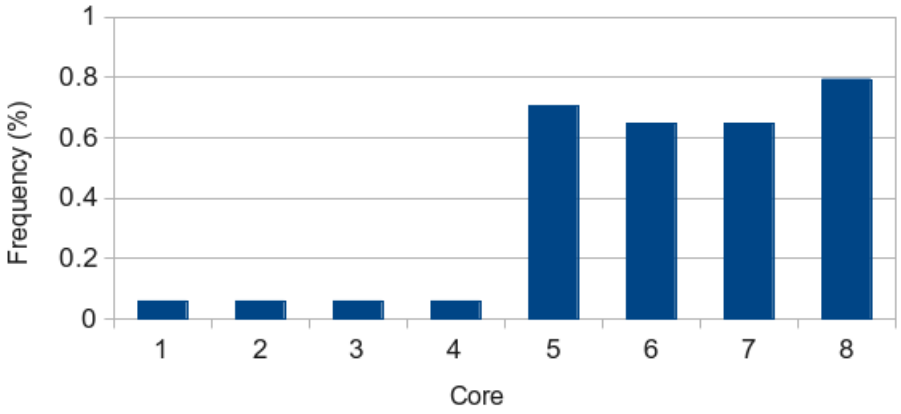

Figure 5.8: Frequency of time a given core spends with the lowest ratio of updates sent to updates received for the Blackscholes benchmark

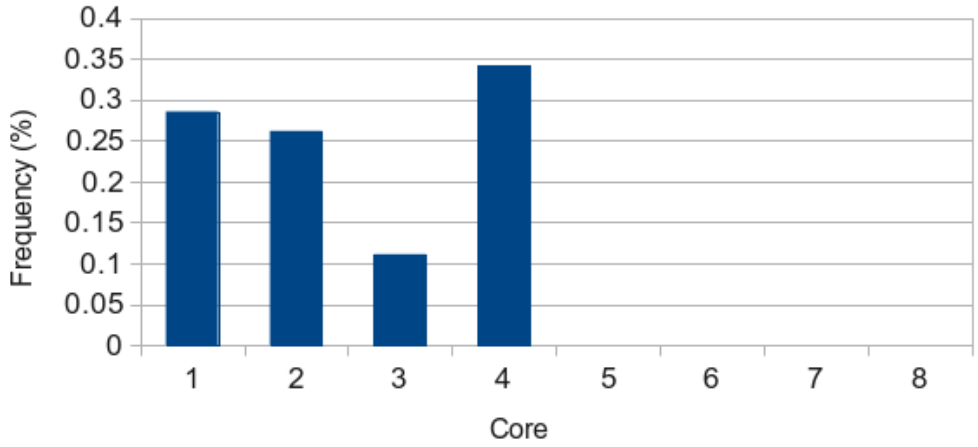

Figure 5.9: Frequency of time a given core spends with the highest ratio of updates sent to updates received for the Fluidanimate benchmark

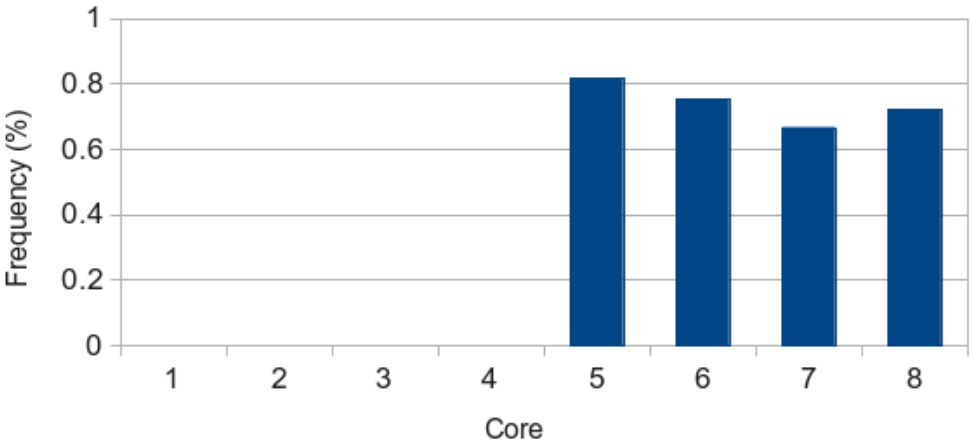

Figure 5.10: Frequency of time a given core spends with the lowest ratio of updates sent to updates received for the Fluidanimate benchmark

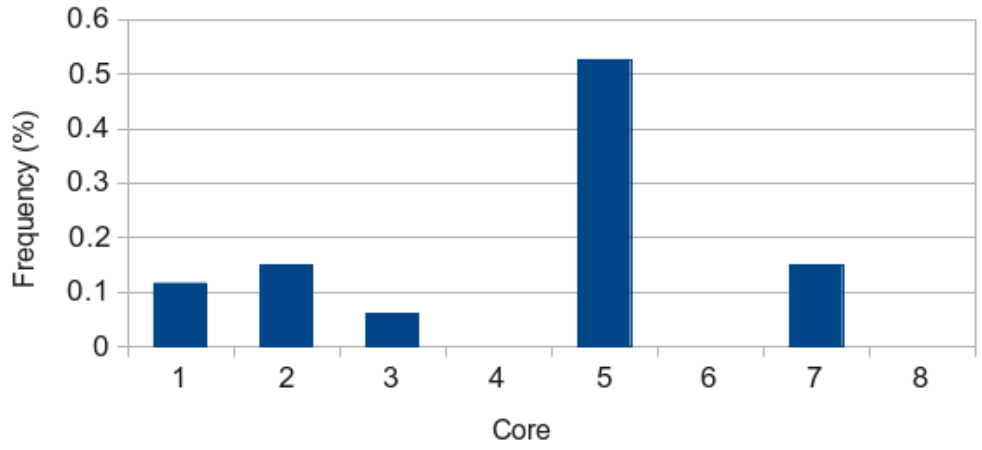

Figure 5.11: Frequency of time a given core spends with the highest ratio of updates sent to updates recieved for the Vips benchmark

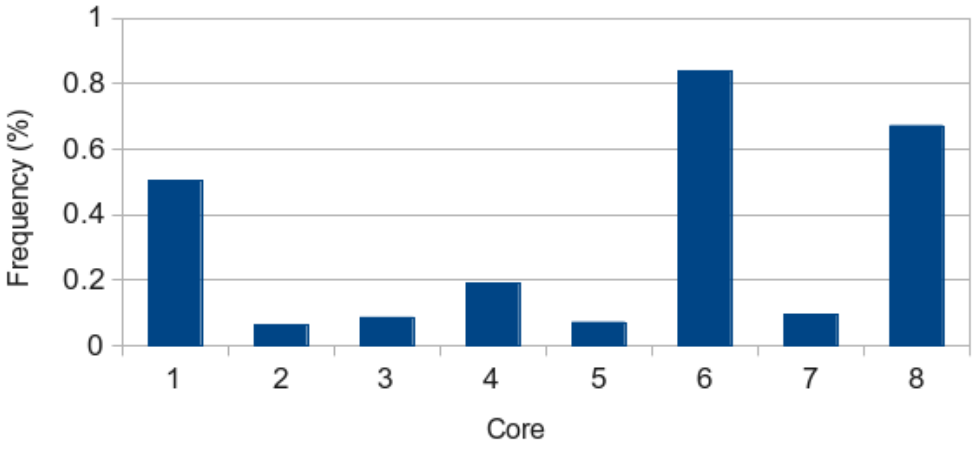

Figure 5.12: Frequency of time a given core spends with the lowest ratio of updates sent to updates recieved for the Vips benchmark

5.6.5 Write Frequency - revisited

In section 5.6.1 we showed the required write frequency presents a problem even in the best case, especially when considered alongside the possibility of a further update from the local predictor. Therefore alternative communications strategies were investigated. The simplest was to reduce the number of updates sent between BPUs. This was achieved by studying a range of different communications schemes.

The Write Always scheme requires the largest number of updates per cycle, with the other schemes taking progressively fewer updates as the aggressiveness of reduction in sharing is increased. The Write Miss scheme only sends updates to other cores

when the local core has mispredicted, the understanding being that while a misprediction generally has some information to be shared, a correct prediction often does not. The Write Reduced scheme sends updates for a limited range of circumstances, namely only updating counters updated by the core's own local update scheme.

In section 5.6.2 conditional direct branches were observed to be most suitable for our scheme. As a result we created the Write Conditional scheme where only updates for conditional branches are shared.

Two further schemes were added. The Write Slipstream scheme makes use of the ratio of updates sent to updates received introduced in section 5.6.4. The same intuition about a core with a higher ratio being ahead and a core with a lower ratio being behind to allow us to only send updates backwards. The exception was that this should reduce less useful updates being sent from the cores which are furthest behind to those furthest in front. Finally the Write Dynamic technique uses thresholds on the number of mispredictions in a given number of cycles to decide whether to share updates or not.

The same benchmarks were re-run and data collected in the same fashion as in section 5.6.1. Looking at figure 5.13 we see a comparison of what happens to the updates per cycle required under the initial Write Always scheme compared the new schemes.

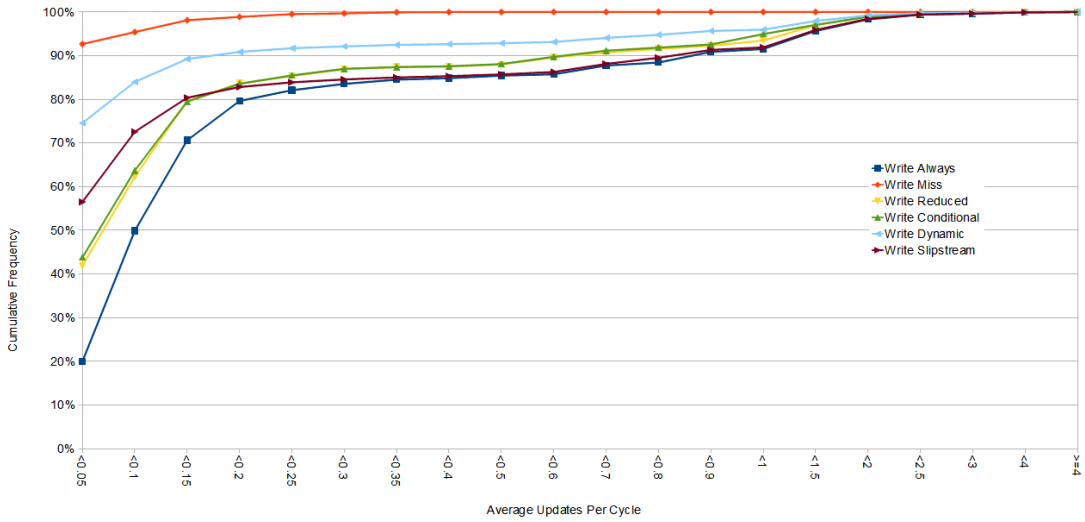

Figure 5.13: Remote updates per cycle versus what portion of the time all remote updates can be accepted. Results are collected in the same way as for figure 5.4.

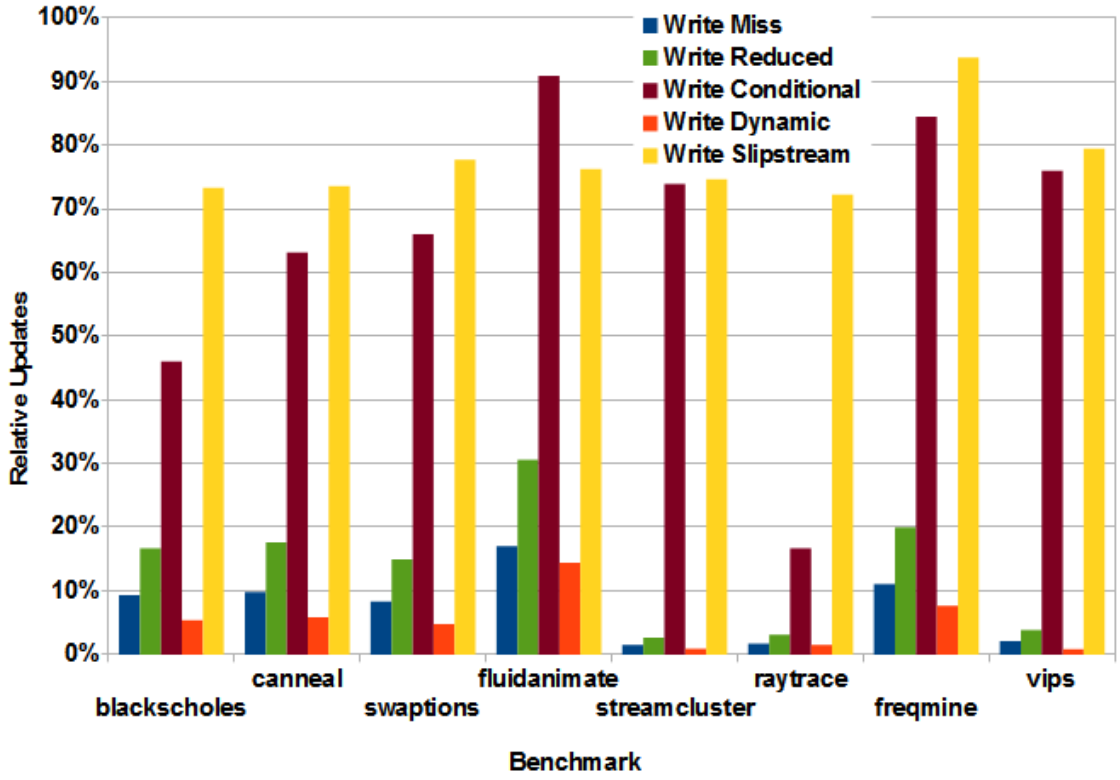

Figure 5.14: Number of updates sent between BPUs per benchmark relative to Write Always.

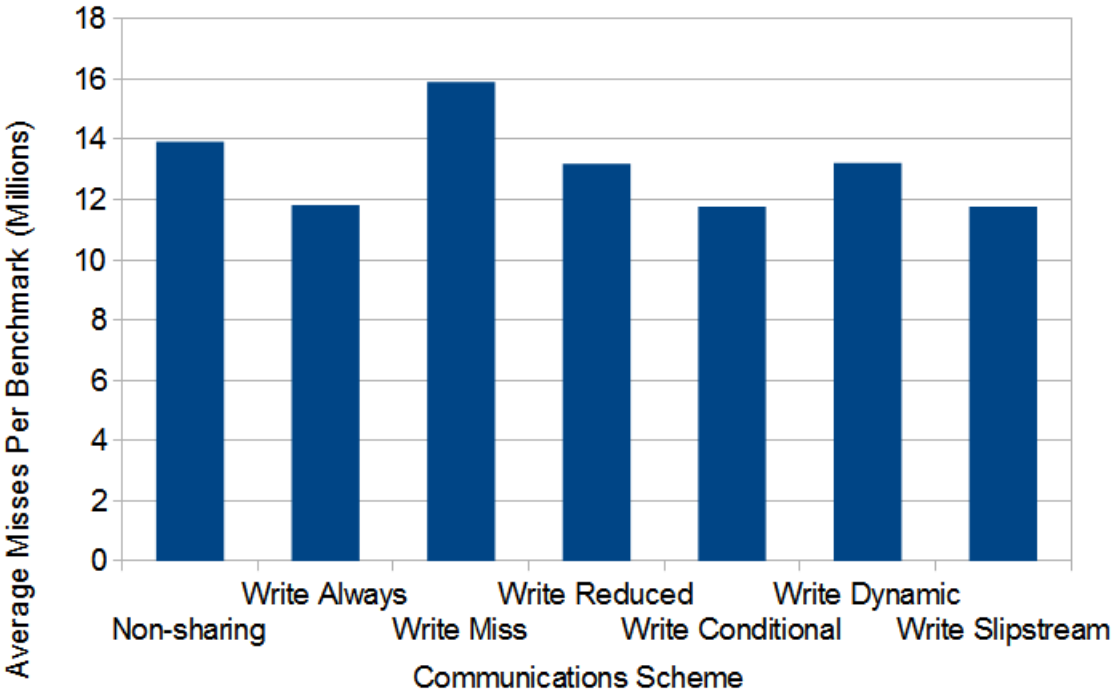

Figure 5.15: Average mispredictions per benchmark for each communications scheme.

Figure 5.14 shows how many updates the new communications schemes send compared to the Write Always scheme. Write Slipstream reduces the number of updates by around 30%, while Write Dynamic Cut-off and Write Slipstream reduce updates by 80-90%.

Figure 5.15 shows that the new Write Conditional and Write Slipstream schemes do not reduce the improvement in misprediction rate over non-sharing, however the Write Miss and Write Reduced schemes do.

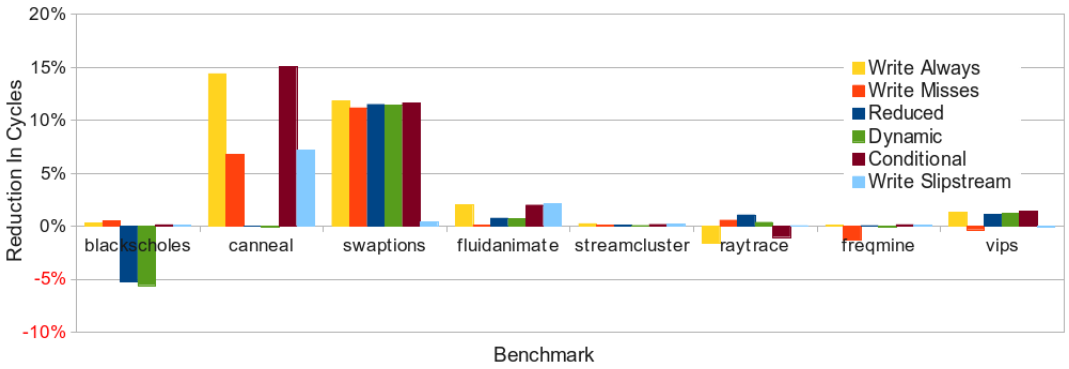

Figure 5.16: Improvement in cycles per benchmark for each communications scheme compared to a baseline of sending no updates between predictors.

Figure 5.16 shows that for most of the benchmarks there is little impact, only around 1-2% speed-up. However, in Blackscholes the Write Reduced and Write Dynamic schemes increase cycles by 5%. The best performance is seen in Canneal, with a speedup of up to 15%, closely followed by Swaptions with a speedup of around 12%.

5.7 Communications Implementation

5.7.1 Software

In order to communicate the updates between BPUs it is necessary to have some physical connection. We chose to model this as a two way connection between the interconnect and the BPU located on each core. This gives the lowest overhead in terms of additional hardware requirements and complexity. However, this will result in increased contention, leading to a possible slowdown. The updates were transmitted by broadcasting to all other cores attached to the data bus.

5.7.2 Hardware

The model provided by MARSS is of a split data/address bus, with an arbitration latency of 1 cycle and a broadcast latency of 6 cycles. We model the addition of the BPUs through the addition of new BPU controllers that attach to the split bus in the same way as the L1 data/instruction caches. In this manner the BPUs communicate in much the same way as any other cache connected to the interconnect, but only interact with other BPUs.

5.7.3 Data transmitted

The data to be sent in each update depends on the type of BPU in use. The information transmitted on a broadcast update, along with the bits cost for each field, is shown in table 5.2. The information included in each update is designed to match the information required by a the local predictor updating its own BPU in the same manner as a standard single core BPU update. Each update, be it a hit or miss, includes all the information required to update each of the BPU structures as necessary according to the state of the BPU receiving the update and the standard BPU update mechanism.

Information Sent	Bits
GShare Index	16
Bimodal Index	16
Meta Index	16
BTB Index	10
BTB Index	21
Target	31
Taken?	1
Total	111

Table 5.2: Information broadcast in an update along with the bits used

It is assumed that a predictor can update its counters on the cycle the update arrives (for any number of remote updates) and that this does not interfere with reads or writes to the predictor from the local core. This is based upon the results from figure 5.13, which showed that nearly 90% of the remote updates could be handled by updating every other cycle. Taken with the fact that the BPU will not have local updates anywhere near this frequently, the addition of a single entry remote write buffer should be sufficient to capture the vast majority of remote updates.

5.8 Key Results

Given the experiments covered in section 5.6 our best configuration proved to be Write Conditional. Applying this to the original configuration used in our limit study gives us a realistic implementation, with the results shown in figures 5.17 and 5.18. These results differ from those in section 5.6.5 in that the impact of transmission latency and bus contention are now taken into account.

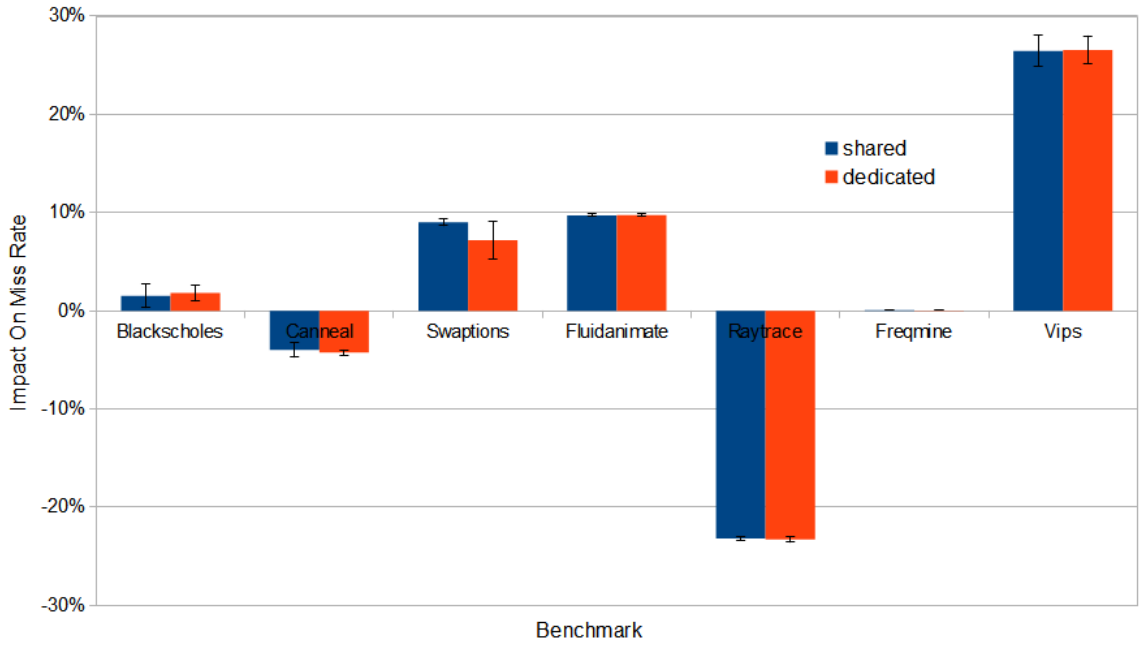

Figure 5.17: Demonstrating the effect of Peloton branch prediction on a range of data parallel benchmarks. The baseline for each benchmark is running the benchmark without sharing. The two sets of results are for sharing the interconnect with cache traffic and for adding a dedicated bus. A negative number shows an increase in branch mispredictions. The error bars shown are the 95% confidence interval from the standard error

Figure 5.17 shows that while Peloton branch prediction can be used to reduce the miss rate for several of our benchmarks by up to 25%. However, the figure also shows that for the wrong benchmark the application of Peloton branch prediction can also lead to a dramatic decrease in prediction accuracy. Just as bad is the potential for the technique to have an extremely low or statistically insignificant impact on the accuracy for the benchmark. Such a result means that the extra time and energy dedicated to transmitting the data across the interconnect and writing it to other BPUs is essentially wasted and should be avoided. The difference between the shared and dedicated archi-

tectures is either within the margin for error or small enough that the extra complexity and die space required to add a new dedicated interconnect for BPU traffic cannot be justified.

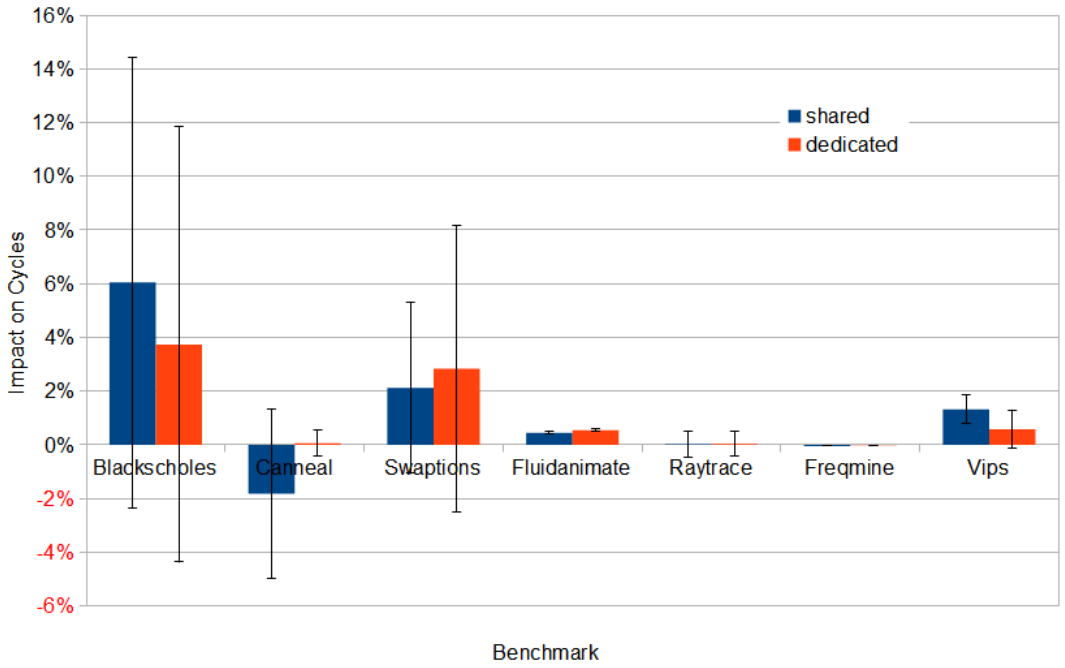

Figure 5.18: These results were obtained in the same manner as figure 5.17 but now concern the impact on program cycles. A negative number shows an increase in program cycles. The error bars shown are the 95% confidence interval from the standard error.

Looking at figure 5.18, the margin for error is greater than the observed impact on cycles for all of the benchmarks except Fluidanimate and the shared interconnect data for Vips. In both cases there is a slight (0.5-1%) reduction in cycles. As a result it appears that there is generally a negligible impact on cycles as a result of applying Peloton branch prediction.

When taking figures 5.17 and 5.18 together it becomes clear that although Peloton branch prediction can have an impact on miss rates this will not guarantee a decrease in cycle counts. Both Vips and Fluidanimate show a large improvement in branch prediction accuracy but a smaller improvement in cycle times, suggesting that while BPU accuracy is an important aspect to the cycles required to run the benchmark, the benchmarks are not bounded by the BPU accuracy.

Figure 5.19 shows the energy required to run each of the benchmarks, both for the sharing and dedicated interconnect architectures. The figures were obtained from

a modified version of the Xeon model that comes with McPAT, but the fraction of energy saved using the ARM A9 model was highly similar. The xml input file for McPAT was populated by the values output by MARSSx86 to ensure that the energy figures given are accurate.

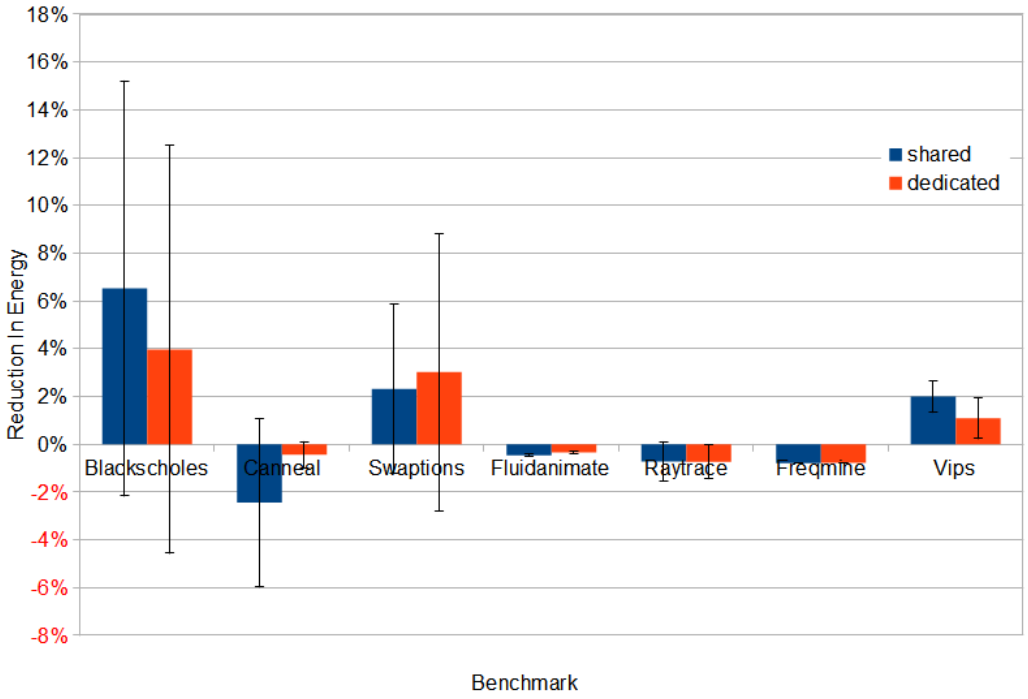

Figure 5.19: Reduction in energy per benchmark for shared and dedicated interconnect architectures. The baseline for each benchmark is running without sharing. Negative numbers mean an increase in energy required to run the benchmark.

The error bars in figure 5.19 show that the energy required to run Blackscholes, Canneal, Swaptions and Raytrace varies between executions, making it difficult to draw any hard conclusions from the data here. The overall shape of figure 5.19 is very close to that of 5.18, demonstrating that the runtime of the benchmarks is the largest factor in the energy consumed. However, the results for Fluidanimate in figure 5.18 show a slight speed up, whereas the results in figure 5.19 show an increase in energy requirement. This suggests that while some energy is saved from the reduction in runtime, the energy required by the interconnect to transmit the BPU updates is large enough to result in an overall increase in energy required. The best result is for benchmark Vips.

The energy required to run Vips is reduced by around 2% for the shared interconnect and 1% for the dedicated interconnect. Once again, the performance of the shared interconnect vs the dedicated interconnect follows the same pattern as the runtime of

the benchmark. As the dynamic energy required to transmit the BPU updates is the same under both schemes the difference between energy will come down to the extra static energy required by an additional interconnect and the change in runtime of the application. This result reinforces the conclusion that a dedicated interconnect for BPU updates is not desirable.

5.9 Further Evaluation

In this section the effectiveness of Peloton branch prediction is further explored. First a limit on the maximum break-even transmission energy is presented. Next the design space of different predictor sizes is presented. Finally a comparison to the Slipstream processor architecture is made.

5.9.1 Energy

A major factor in whether or not Peloton branch prediction will consume more or less energy than is required to run the same system without sharing branch updates is the energy consumed in sending updates between the BPUs. This section estimates an approximate upper bound for the energy that can be consumed by looking at the mispredictions removed by using Peloton branch prediction and calculating the cycles saved as a result. This is balanced against the number of updates sent to give an upper bound on the cost of each update for total energy to remain the same.

$$Misses_{Saved} \times Penalty_{Misses} \times Power_{Core} > Transmission_{Cost} \times Transmission_{UpdatesRequired} \quad (5.1)$$

The Atom core used in section 5.8 was plugged into McPAT and found to consume 2.011W. The branch miss penalty is 6 cycles. Using the figures from running Blackscholes on the 64K shared BPU configuration gives 96723 misses saved and 16.5 million updates sent. From equation 5.1 this gives:

$$96723 \times 6 \times 2.011 > Transmission_{Cost} \times 16.5x10^{6} \quad (5.2)$$

$$Transmission_{Cost} < 70\text{mW} \quad (5.3)$$

Given that the peak dynamic power for the NoC reported by McPAT is 66mW we can start to see why Peloton branch prediction may be able to provide small power and energy savings.

5.9.2 Design Space Exploration

Varying GShare BHT, Metatable and Bimodal entries from 2,048 to 65,536, the BTB sets from 512 to 1024 and setting BTB ways to 2 or 4 gives the results shown in figures 5.20 and 5.21.

These figures show that Peloton branch prediction performs slightly worse for smaller predictor sizes. This is especially true of Vips, with only the largest BPU resulting in a significantly improved miss rate and reduction in cycles. This is likely to be due to an increase in aliasing at the smaller predictor size due to the fewer table entries being unable to cope with the increase in updates caused by updates from other cores.

The results for Swaptions are very interesting. All of the BPU sizes resulted in an increase in cycles, with the smaller BPU sizes resulting in a smaller increase. However, the opposite is true of the impact on miss rate, with all but the smallest two BPU sizes showing a significant increase in accuracy.

Raytrace also shows some interesting results. Each predictor size shows a reduction in accuracy but no significant change in the number of cycles. This would suggest one of two things: either the impact of BPU accuracy is not a limiting factor in the performance of the benchmark (the baseline miss rate is only a little over 1%, making this quite likely); or that any reduction in cycles achieved by the increased BPU accuracy is negated by the extra traffic generated on the shared bus, making memory, instruction cache and data cache accesses slower.

Figure 5.22 shows the impact on energy savings as the BPU size is reduced. The overall picture is that Vips is the only benchmark to show a definite benefit from sharing updates, and only then for large predictor sizes. The reason for this is likely to be that as the BPU size is reduced the energy that it consumes is reduced, so the fraction of total processor energy that may be saved is reduced. When taken with the increasing number of branch mispredictions and the corresponding increase in BPU updates the energy consumed in the additional cycles and interconnect traffic may become a proportionally larger factor than the branch mispredictions that are prevented through BPU updates.

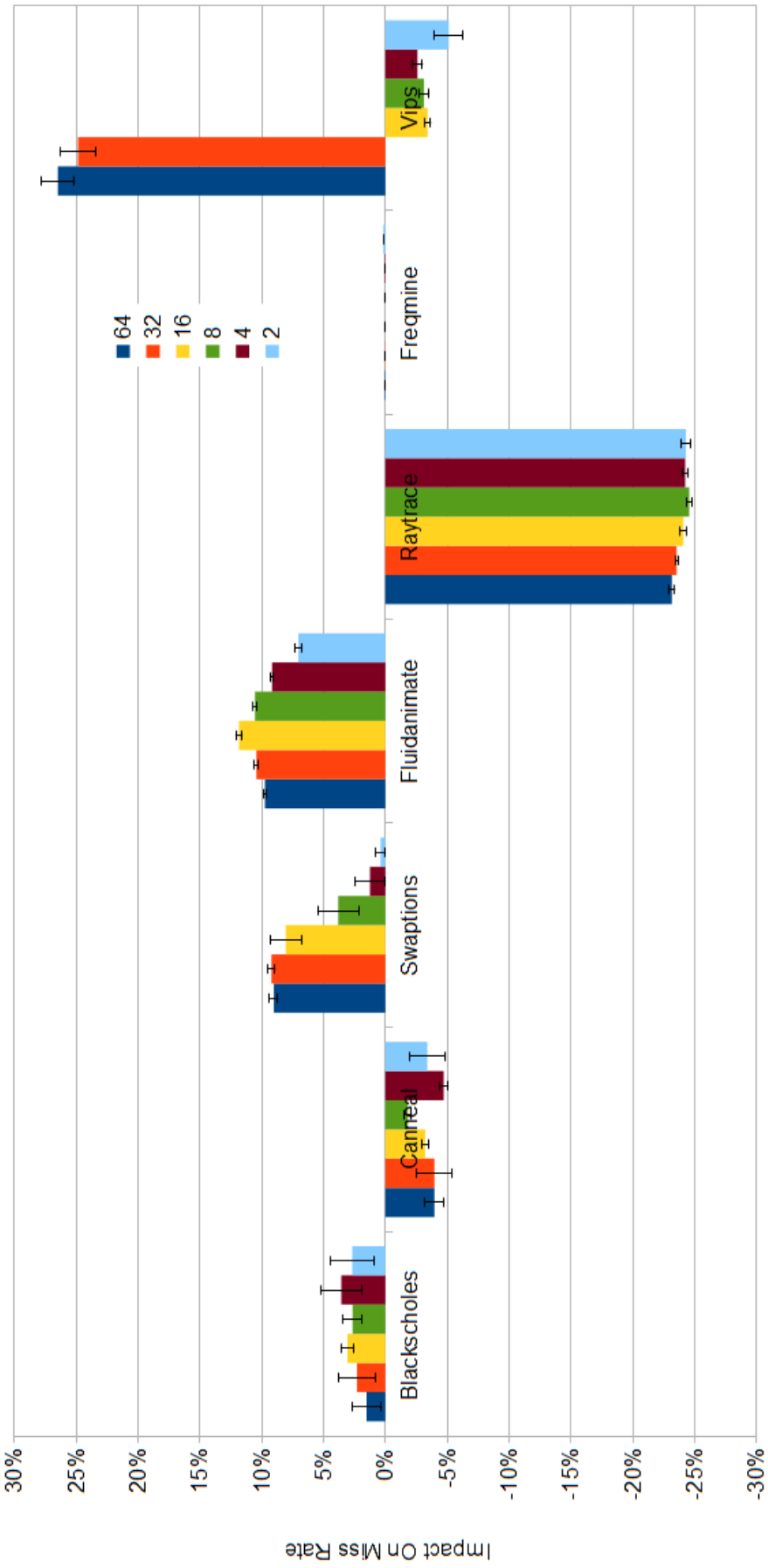

Figure 5.20: Demonstrating the effect of Peloton branch prediction on branch predictor accuracy for a range of data parallel benchmarks over a baseline of not sharing updates. The different sets of results are for different sized branch predictors, from 64K to 2K entries in each of the direction predictor subtables. A negative number shows an increase in mispredictions.

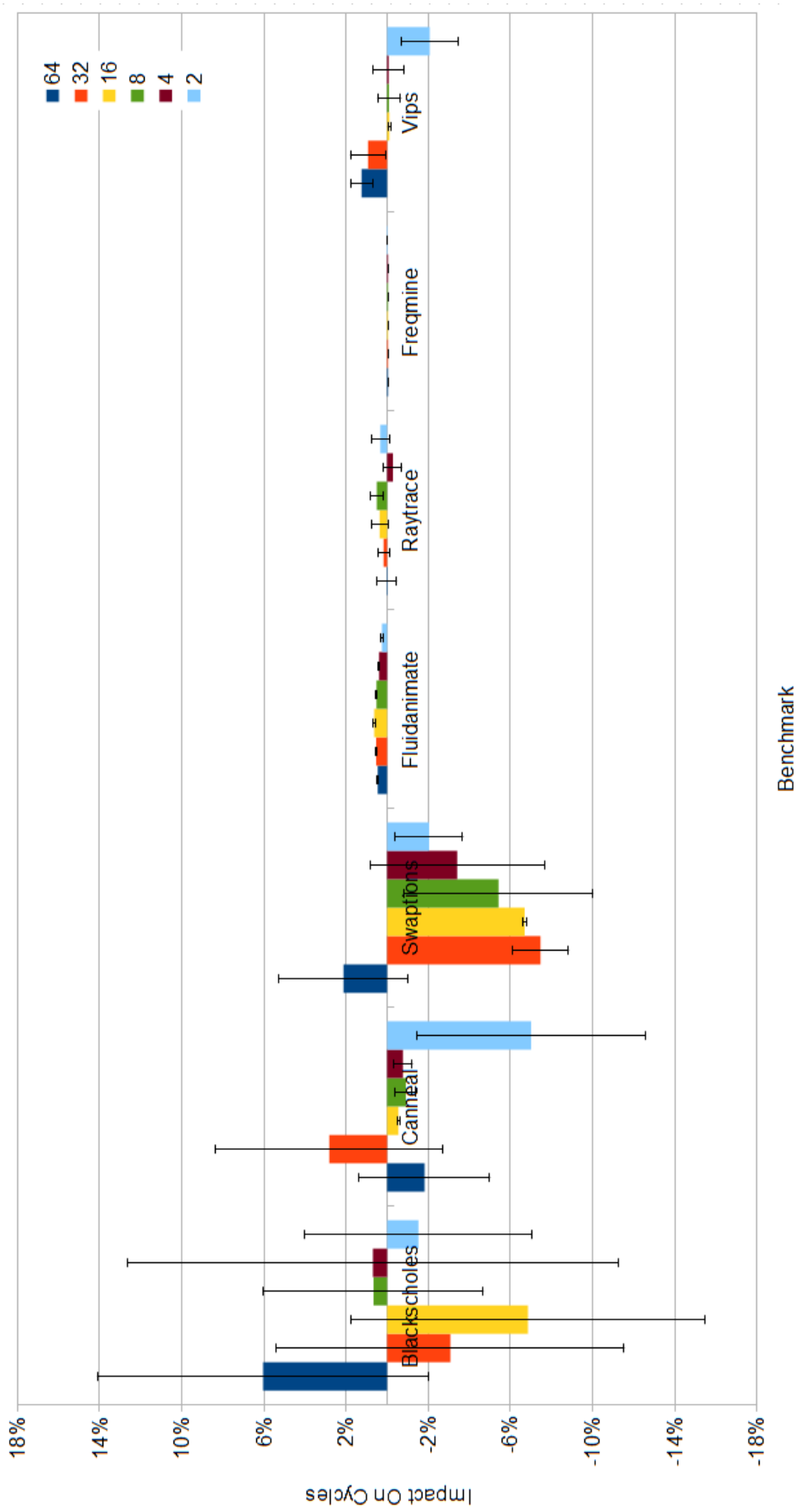

Figure 5.21: Demonstrating the effect of Peloton branch prediction on cycle count for a range of data parallel benchmarks over a baseline of not sharing updates. The different sets of results are for different sized branch predictors, from 64K to 2K entries in each of the direction predictor subtables. A negative number shows an increase in cycles.

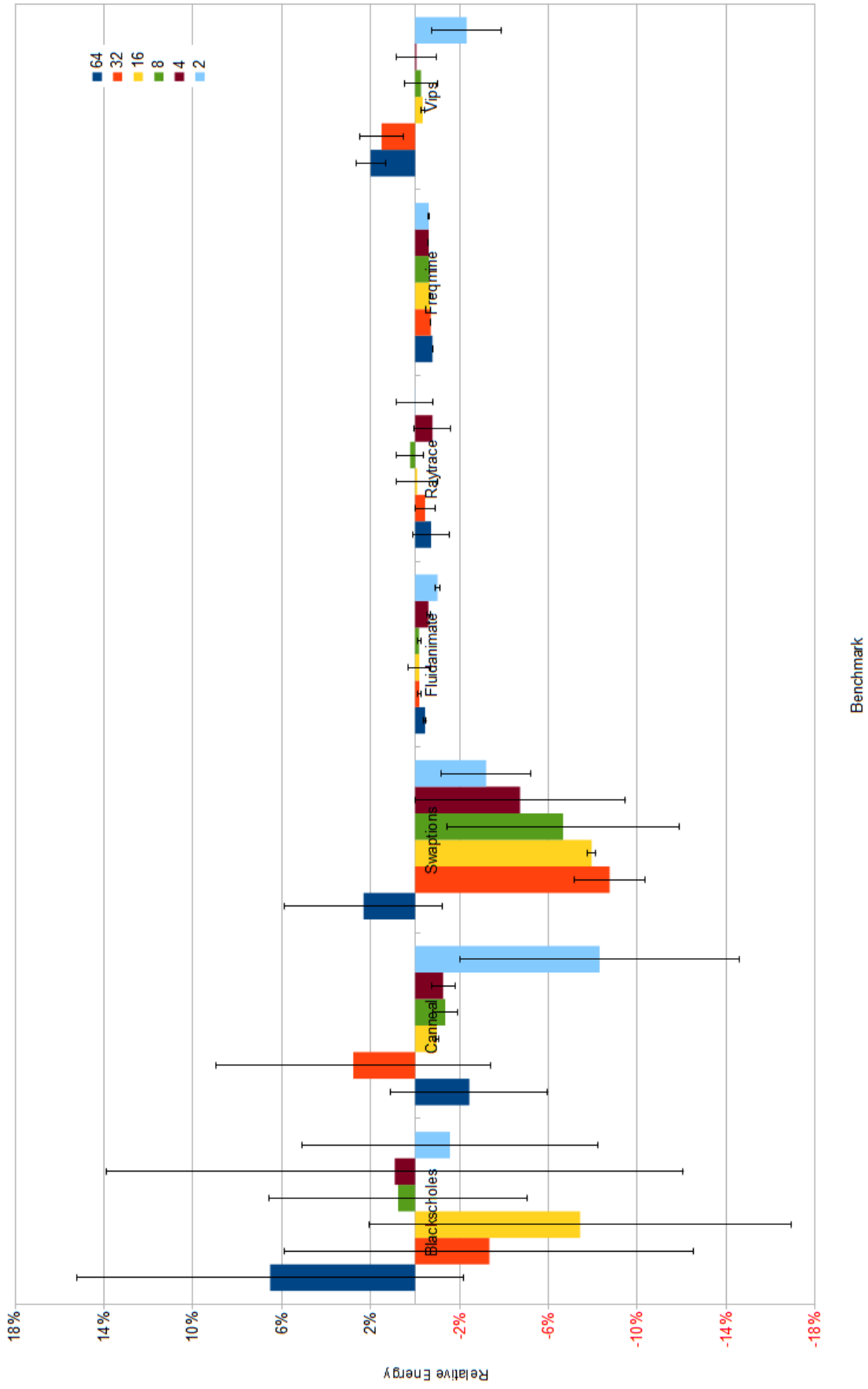

Figure 5.22: Demonstrating the effect of Peloton branch prediction on energy required to complete a range of data parallel benchmarks over a baseline of not sharing updates. The different sets of results are for different sized branch predictors, from 64K to 2K entries in each of the direction predictor subtables. A negative number shows an increase in energy.

5.9.3 Comparison To Slipstream Processors

The Slipstream Processor described in [65] differs from the processor described in this chapter in several ways, most importantly that it models an out of order core and an ideal L2 cache which always hits. In [65] the SPEC95 integer benchmark suite, although the instruction count given is similar to the runtime of the benchmarks used here.

The Slipstream Processor manages an average improvement of 7% increase in performance, however this goes as high as 20% and as low as no improvement, with no reported slowdown. This means at in the worse case an entire processor is being used to run the reduced thread and producing no speed-up, unless it is known ahead of time that the application will not benefit from slipstreaming. In contrast, Peloton branch prediction produces up to 1% speed-up across each of the 8 processors, with no need for a whole processor dedicated to overhead.

As a result, Peloton branch prediction is likely to be more energy efficient and is applicable in a greater range of circumstances. This can generally be attributed to the differing goals of the two techniques: the slipstream processor seeks to make use of an otherwise unused core to produce a speed-up in a single application at all costs, whereas Peloton branch prediction aims to maintain performance while reducing energy and die-space requirements.

5.10 Other Predictor Types

The most successful technique from section 5.8, Write Conditional, was also applied to the other base predictor types identified in section 5.4.

5.10.1 L-TAGE

The L-TAGE predictor is based on the solution provided to the second branch predictor championship [33]. The predictor is comprised of 12 partially tagged components with history lengths ranging from 4 to 640 and consisting of between 2048 and 512 entries, along with a 16K entry bimodal base predictor and a 256-entry loop predictor (see [61] for full implementation details).

The L-TAGE predictor updates comprise the TAGE sub-predictor containing the longest entry, the TAGE index, the BTB index and tag, the outcome of the branch and the branch target, for a total of 78 bits.

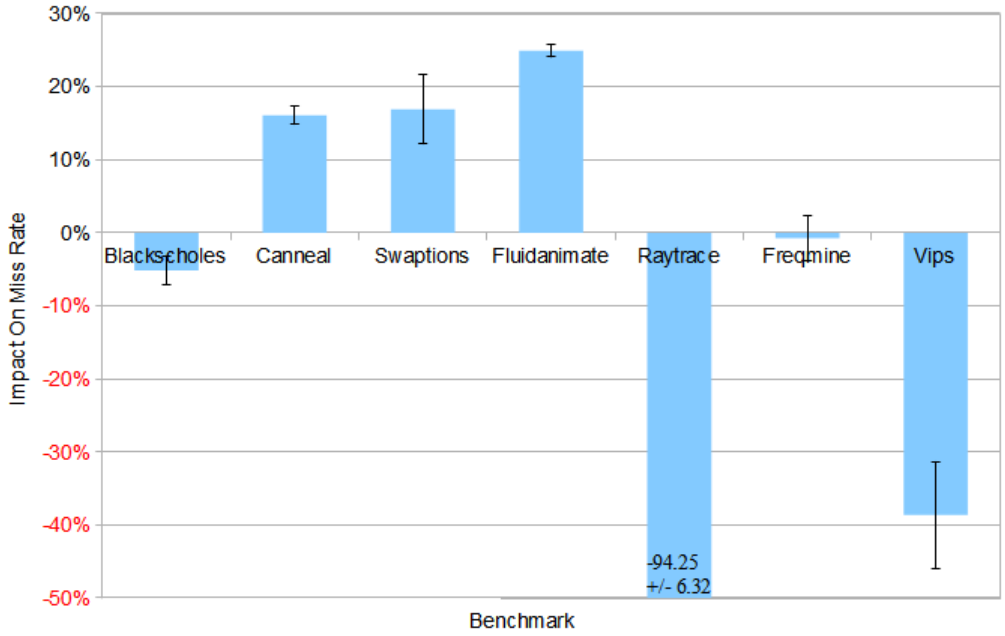

Figure 5.23: Impact on miss rates from applying Peloton branch prediction to the L-TAGE predictor for a shared interconnect. Baseline is L-TAGE predictor without shared updates.

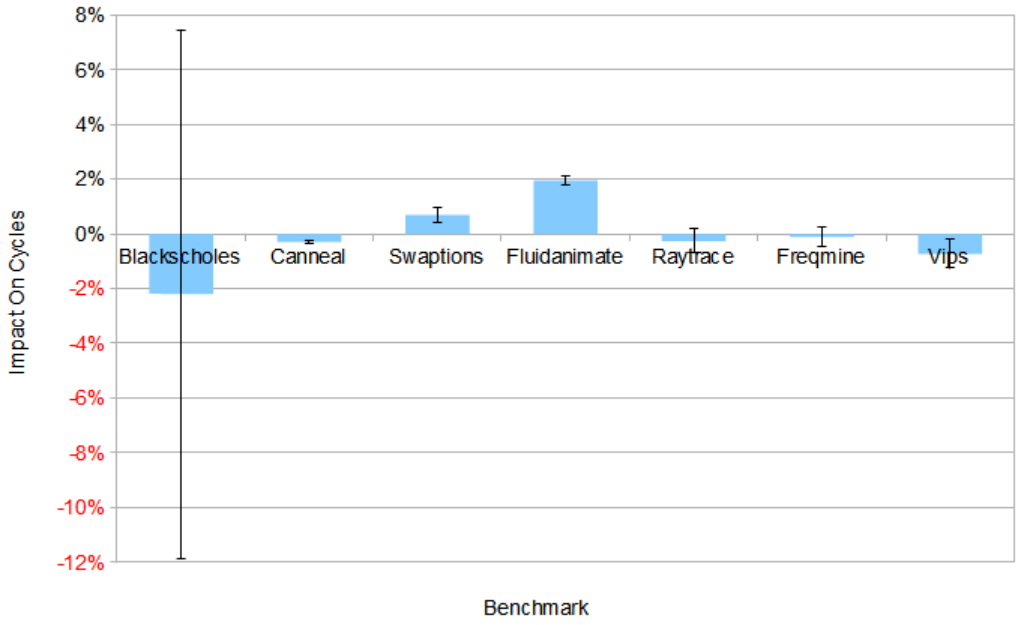

Figure 5.24: Impact on cycles from applying Peloton branch prediction to the L-TAGE predictor for a shared interconnect. Baseline is L-TAGE predictor without shared updates.

Benchmark	Unshared Miss Rate	Shared Miss Rate	Unshared Cycles (Millions)	Shared Cycles (Millions)
Blackscholes	8.16% (± 0.06)	8.58% (± 0.15)	333 (± 15.28)	340 (± 28.57)
Canneal	8.78% (± 0.08)	7.36% (± 0.07)	342 (± 0.04)	343 (± 0.10)
Swaptions	8.53% (± 0.35)	7.09% (± 0.18)	735 (± 0.86)	729 (± 1.62)
Fluidanimate	22.69% (± 0.19)	17.03% (± 0.02)	1,296 (± 1.84)	1,271 (± 0.51)
Raytrace	1.80% (± 0.10)	3.50% (± 0.02)	2,469 (± 7.60)	2,476 (± 7.50)
Freqmine	12.86% (± 0.33)	12.95% (± 0.22)	4,424 (± 12.73)	4,429 (± 8.31)
Vips	3.40% (± 0.24)	4.71% (± 0.05)	2,688 (± 5.79)	2,708 (± 13.08)

Table 5.3: Miss rates and cycle counts for TAGE predictor with and without update sharing

The results in figure 5.23 show that sharing BPU updates can have a large, positive impact on branch predictor accuracy in several of the benchmarks. However, much as with the GShare predictor, sharing updates can also result in a large increase in mispredictions. It is worth noting that while Raytrace and Vips both had a large negative response to sharing updates they had the lowest misprediction rate to start with (1.8% and 3.4% respectively). This explains why figure 5.24 shows only a small increase in runtime for these two benchmarks.

The best results are shown to be for Swpations and Fluidanimate, where mispredictions are reduced by 15-25% and runtime reduced by 1-2%. This shows that in the right circumstances sharing updates can lead to an improvement in branch predictor accuracy and a reduction in runtime for not just hybrid GShare predictors, but state of the art TAGE predictors as well.

5.11 Further Work

With modern multi-core processors heterogeneous cores are an increasingly popular feature. In this chapter we limited ourselves to homogeneous cores with identical branch predictors as this made transmitting updates between BPUs much easier. It should be possible to share updates between differently sized BPUs, but this will require an extra step or transferring extra data due to the different hashing functions that would be applied in different sized BPUs.

It would also be worth looking at adding extra logic to be able to handle the case where more than one application is running. This is because, without the correct protocol being in place, it would be easy for the updates for one application to overrate the counters for the other application, resulting in reduced performance. This technique would require some mechanism for identifying when different virtual addresses are mapping to the same source code branch.

5.12 Summary

An early assumption of this chapter was that it is possible to exploit the similarities in branch outcomes across multiple cores executing the same data-parallel application on different data points. The results prove that this assumption holds true. The results have shown a reduction in misprediction rate of between 1% and 25%, accompanied by a reduction in run time of a between 1% and 6%.

The energy model based on McPAT showed a reduction in energy of 1% and 6%. The approach of making use of the existing split address/data bus by simply connecting it to the BPUs means that the solution introduces a minimum of hardware complexity.

This chapter has demonstrated that Peloton branch prediction is suitable for a range of different branch predictor sizes, and provided an explanation as to how and why it is successful. Finally, Peloton branch prediction has been shown to be effective not only for GShare based hybrid predictors, but also cutting edge L-TAGE predictors.

Chapter 6

The Case For Heterogenous Cooperative Branch Prediction

6.1 Introduction

In chapter 5 a new technique for the improvement of performance in a multicore data-parallel setting through better BPU performance was introduced. This technique looked at sharing branch predictions between the BPUs that are traditionally found as part of a multicore BPU, with the addition of a connection to the interconnect. The results showed that by passing updates between the BPUs accuracy could be improved and a runtime speedup obtained.

The movement towards multicore processor design was highlighted in chapter 1. This chapter went on to explain the growing interest in processors employing a heterogeneous core selection, where processor cores with different computational power and energy requirements are employed to meet a range of differing application-run-time requirements.

In this chapter we consider the application of our novel BPU communication technique to a heterogeneous processor design. In this case the aim is to use a selection of cores with smaller BPUs and some with larger BPUs with the aim of achieving similar, or even better, performance than can be achieved by using only the larger BPUs. If this performance goal can be met then it will be possible to achieve a lower cost, lower energy design requiring less die space but with the same runtime.

The rest of the chapter looks at a more detailed motivating example of what a heterogeneous design can achieve, before moving on to a detailed discussion of heterogeneous design simulated in our experiments. The results of our simulated design

are then presented, followed by an analysis of what they mean for performance and BPU sizes. Finally we consider further work in the area and then present a summary of the chapter.

6.2 Motivation

The advantages of a smaller BPU were discussed in chapter 1 but can be summarised as resulting in lower energy consumption, smaller die-space and a cheaper chip. A reduction in BPU size will generally result in a reduction in prediction accuracy, but there are some cases where the design is constrained by either cost, energy or space, leading processor architects to consider a processor with a smaller BPU. Through the use of our novel communication technique, we seek to be able to reduce the size of some of the BPUs without sacrificing the performance of the overall system.

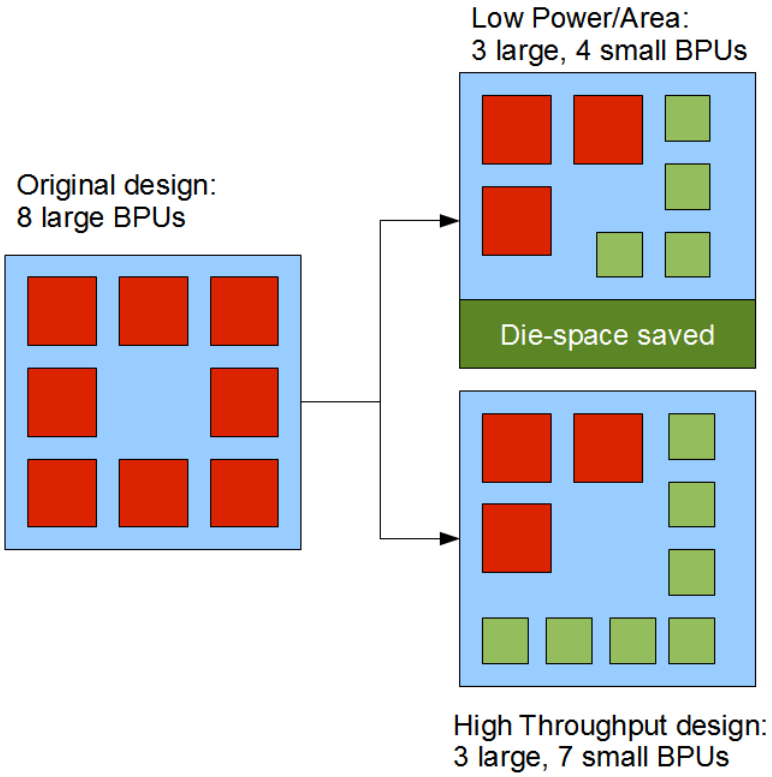

Figure 6.1: Demonstrating two ways that heterogeneous BPU sizes can impact on processor design.

In chapter 5, our analysis of the cyclic slipstreaming behaviour of the processor cores suggested that not all of the cores were doing an equal share of work as the first core to encounter a branch. As a result it may be possible to have a small number

of cores with large BPUs that would do the majority of the hard work in predicting branches before passing the branch information back to the following cores which would only need a small BPU to turn this extra prediction information into a more accurate prediction. In an ideal case the following cores with the smaller BPUs would need only a handful of BPU entries to accurately predict all the branches they encountered, removing the large energy and die-space requirements that were present before.

This chapter considers the application of the technique to data-parallel benchmarks as these should maximise the opportunity for branch information to be shared between BPUs. The reduction in energy afforded by this technique could be used by large data farms to directly reduce the energy consumed by their processors and thus reduce their energy bill. Furthermore, running the cores with lower power and energy constraints could result in reduced thermal load, allowing for a reduction in cooling and further savings. Alternatively, the same energy and die-space could be used to add an additional processor core with the small BPU, thus resulting in increased throughput and a reduction in time required to process a data set.

Large Predictors	Small Predictors	Saving at Half Size	Saving at Quarter Size
6	2	12.5%	18.75%
4	4	25%	37.50%
2	6	37.5%	56.25%

Table 6.1: Savings to bits budget by using a combination of full and half or quarter sized predictors.

If the bits budget of the BPU is 20% of the overall processor budget, then reducing 6 of the BPUs to 25% their original size will have reduced the bits budget sufficiently to add a 9th core with a similarly reduced BPU and still require a smaller bits budget than the original 8.

6.3 Methodology

In this section we present the architecture of the processor that was simulated when conducting our experiments. The basic set-up is much the same as in chapter 5. The experiments were conducted using the same MARSS simulator [52] (see section 5.4).

We chose to simulate the PARSEC2 benchmark suite [6], containing examples

of demanding, data-parallel workloads, suitable for research into diverse, non-high performance computing applications. We used the simsmall dataset to simulate only the benchmarks described as data-parallel. We used the built in Intel Atom processor model as a representative model of an in-order core tested against real hardware [43].

The modelled core is single threaded, with 2-wide fetch and issue width, 2 integer, 2 floating point and 2 complex functional units, 32-entry commit buffer and 16-entry dispatch queue and store buffer. Each core has a 256-set, 8-way MESI instruction cache and an identical data cache. A single 2^{12}-set, 8-way L2 cache is shared amongst the cores. The benchmarks were each set to run 4 threads over the 8 single threaded cores. The branch misprediction cost was 6 cycles.

The branch predictor modelled comprised a 2^{10}-entry, 4-way BTB, used to predict branch targets and a 2^{10}-entry RAS used for storing call return addresses. The RAS was implemented as a circular stack, with each of the 2^{10} entries storing information about a call-return pair. Each time a call is made a new return address is pushed onto the top of the stack. The number of entries is chosen to be far larger than is likely to be needed to ensure that the performance of the RAS does not hinder the overall accuracy of the BPU. The major effort of the new technique in this chapter is centred around the BTAC and especially the BHT. Each of the RAS entries comprises the 31 bits required to capture the branch return address.

The direction prediction component was a hybrid GShare-Bimodal predictor with a bits budget ranging from 48 KB to 1.5 KB (for details see table 6.3). The hybrid consisted of two sub-predictors (GShare and Bimodal) and meta predictor. The GShare predictor uses a global branch history register to store the outcome of all branches, which is then XORed with the PC of the branch to calculate the index of the 2-bit saturating counter used to predict the branch outcome. The Bimodal sub-predictor uses no history information and indexes its 2-bit saturating counters using the branch PC folded in half by XORing. The meta predictor was implemented in the same way as the Bimodal predictor.

Subpred Entries	Direction Size	BTB size	Ras Size	Total (Per Core)
65,536	16 KB x 3	16 KB	8 KB	72 KB
32,768	8 KB x 3	16 KB	8 KB	40 KB
16,384	4 KB x 3	16 KB	8 KB	28 KB
8,192	2 KB x 3	16 KB	8 KB	22 KB
4,096	1 KB x 3	16 KB	8 KB	19 KB
2,048	0.5 KB x 3	16 KB	8 KB	17.5 KB

Table 6.2: Bits budgets of the BPU components for the different sizes used in this chapter.

The experiments were split into 4 series: the homogeneous architectures where all 8 BPUs are the same size, heterogeneous architectures with 6 large and 2 small BPUs, heterogeneous architectures with 4 large and 4 small BPUs and heterogeneous architectures with 2 large and 6 small BPUs. These configurations were chosen to provide a sampling of the different balance of sizes and still be easily manufactured. For each of the heterogeneous configurations the small BPU's direction predictor tables have half the number of entries found in the larger BPU but are identical in all other respects.

Configuration	Total Die-space (mm^2)	Total BPU KB
8x2048	41.2735	1120
2x4096-6x2048	41.6762	1144
4x4096-4x2048	42.0789	1168
6x4096-2x2048	42.4816	1192
8x4096	42.8843	1216
2x8192-6x4096	43.6914	1264
4x8192-4x4096	44.4985	1312
6x8192-2x4096	45.3057	1360
8x8192	46.1128	1408
2x16384-6x8192	47.7055	1632
4x16384-4x8192	49.2982	1856
6x16384-2x8192	50.8909	2080
8x16384	52.4836	2304
2x32768-6x16384	55.3551	2496
4x32768-4x16384	58.2266	2688
6x32768-2x16384	61.0980	2880
8x32768	63.9694	3072
2x65536-6x32768	69.9425	3712
4x65536-4x32768	75.9154	4352
6x65536-2x32768	81.8883	4992
8x65536	87.8611	5632

Table 6.3: Bits budgets and die-space consumed by the different BPU configurations used in this chapter.

6.3.1 Software

To communicate the updates between BPUs it is necessary to have some physical connection. We chose to model this as a two way connection between the interconnect and the BPU located on each core. This gives the lowest overhead in terms of additional hardware requirements and complexity. However, this will result in increased contention, leading to a possible slowdown. The updates were transmitted by broadcasting to all other cores attached to the data bus.

6.3.2 Hardware

The model provided by MARSS is of a split data/address bus, with an arbitration latency of 1 cycle and a broadcast latency of 6 cycles. We model the addition of the BPUs through the addition of new BPU controllers that attach to the split bus in the same way as the L1 data/instruction caches. In this manner the BPUs communicate in much the same way as any other cache connected to the interconnect, but only interact with other BPUs.

6.3.3 Data Transmitted

The data to be sent in each update depend on the size of BPU in use. Each update consists of the number of history bits needed for the larger predictor, the outcome of the branch and the branch target. This gives $Log_2(\text{larger sub-predictor size}) + 63$ bits per update, giving a maximum update size of 79 bits.

It is assumed that a predictor can update its counters on the cycle the update arrives (for any number of remote updates) and that this does not interfere with reads or writes to the predictor from the local core.

6.4 Results

In this section we present the results of our experiments with heterogeneous BPU configurations alongside the homogeneous configurations to facilitate comparison. This graph is an extension of the work found in chapter 5, figure 5.20. The Unshared-8 and Shared-8 series in figure 6.2 are the same as the Unshared and Shared in figure 5.20.

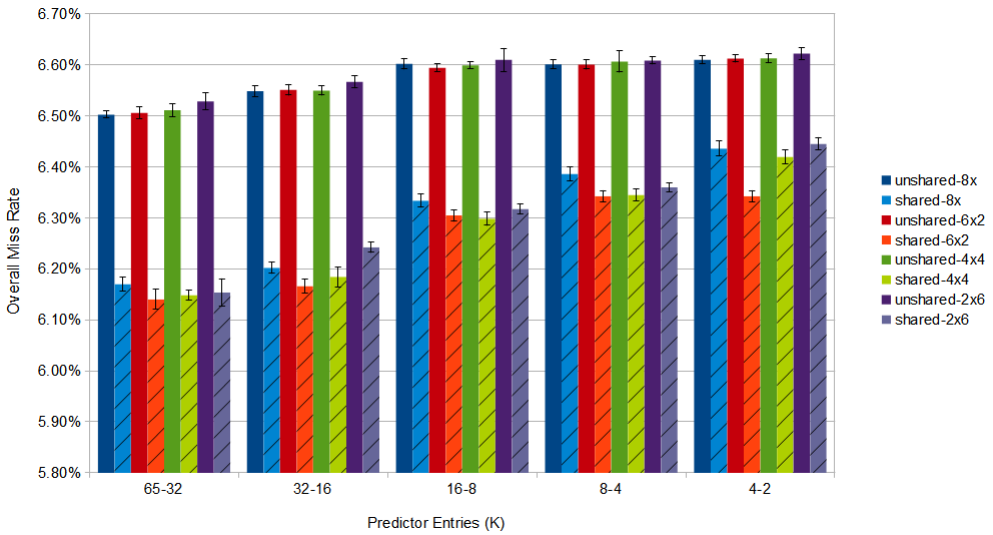

Figure 6.2: Miss rate achieved by finding total mispredictions and completed branches for each of the 5 results per benchmark, then summing over each benchmark and dividing completed branches by mispredictions. Results are given for each processor paring size along the x-axis (smaller predictors to the right) and for each combination of processor pairing. The error bars shown are the 95% confidence interval from the standard error.

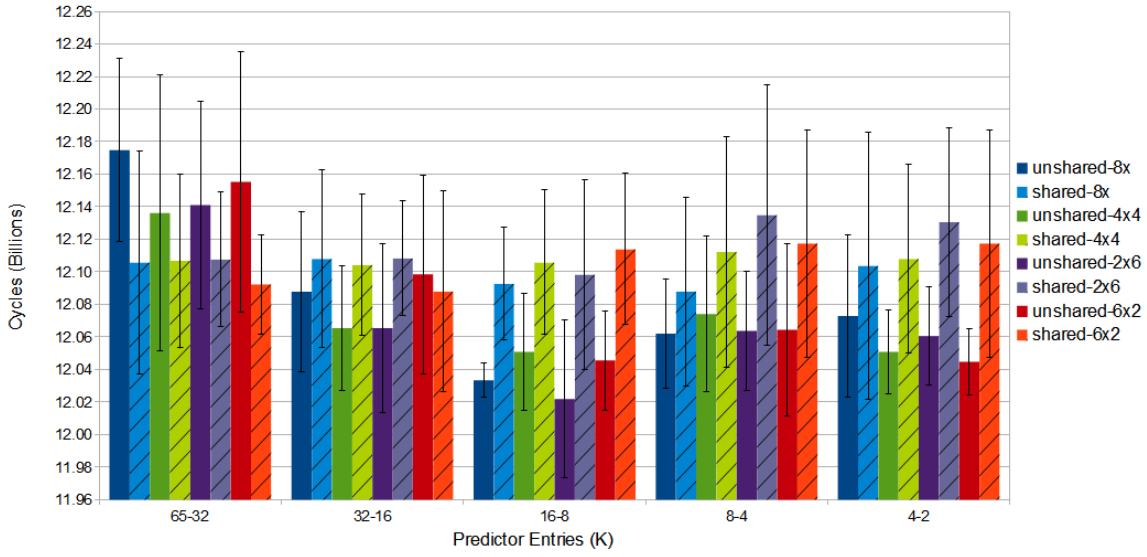

Figure 6.3: Cycles achieved by adding together cycles each of the 5 results per benchmark, then summing over each benchmark. Results are given for each processor paring size along the x-axis (smaller predictors to the right) and for each combination of processor pairing. The error bars shown are the 95% confidence interval from the standard error.

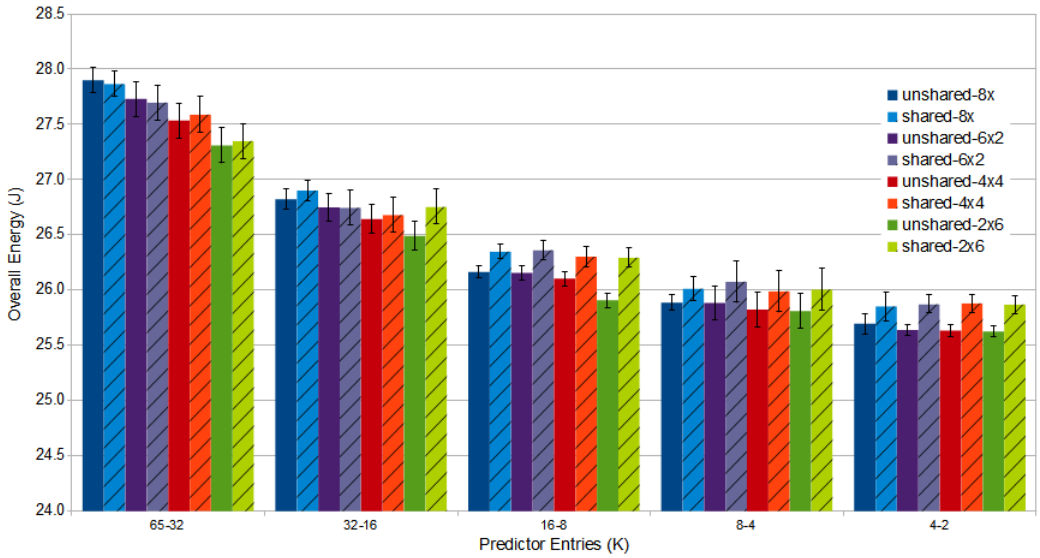

Figure 6.4: Total energy consumed by adding together energy used by each of the 5 results per benchmark, then summing over each benchmark. Results are given for each processor paring size along the x-axis (smaller predictors to the right) and for each combination of processor pairing.The error bars shown are the 95% confidence interval from the standard error.

Figure 6.2 shows how each architectural configuration performs over all benchmarks executed. While the figure generally follows the expected shape of improving performance with increased bits budget while experiencing diminishing returns, there are some interesting results. The most striking is that the results are split into two groups, with the architectures that share BPU information clearly outperforming those that do not. The performance of the architectures making use of sharing is much more sensitive to the bits budget of the BPU than those that do not share updates, with the homogeneous architectures being the most sensitive in both the shared and non-sharing architectures.

An important result to note is that, as was seen in chapter 5, even the smallest configurations with sharing can outperform the largest, most accurate configurations without sharing, achieving an improvement in miss rate of 2%, while reducing the bits budget by around 5x. The most accurate performance was achieved by an architecture sharing updates amongst 6 predictors with 64K entries and 2 predictors with 32K entries. The most efficient (being the a good balance between bits budget and accuracy) is arguably either an architecture sharing updates amongst 6 predictors with 4K entries and 2 predictors with 2K entries, or an architecture sharing updates amongst 4 predictors with 4K entries and 4 predictors with 2K entries. Both predictors perform

only 3% worse than the best shared architecture, 2% better than the best non-sharing architecture and require only 40% of the bits budget. Both architectures outperform a predictor of a similar size that does not use sharing by nearly 4%.

The results in figure 6.3 show the impact changing the architecture has on the cycles required to complete all of the benchmarks. The error bars are much larger than in figure 6.2. This is because the nature of a full system simulator that makes use of randomness will have additional factors that impact the cycle count than just the branch prediction accuracy. By running each benchmark-architecture pairing 5 times the factors that are not due to the branch predictor should be smoothed out as much as possible. It it clear that while sharing does reduce the cycle count for 64K entry configurations, it increases the cycle count in all other cases. Since this is the opposite of what was observed in figure 6.2, it must be concluded that the overhead of the BPU updates across the interconnect outweights the improvement in branch predictor accuracy.

It is interesting to note that the best result in figure 6.3 is found for the 16K entry predictors, with the homogeneous configuration and potentially the 2x16K-6x8K configurations performing the best. This is despite the fact that figure 6.2 shows that these configurations perform worse than the 64K entry configurations. One possible reason for the lower cycle count is that, while the larger configuration made less mispredictions, these may have had a larger secondary cost, such as an increase in I-Cache misses, or may have had a larger apparent cost due to less overlap with other causes of pipeline stalls.

The energy results shown in figure 6.4 very closely mirror the results seen in figure 6.3. All but the largest configurations of the architectures sharing BPU updates consume more energy than those that do not. This will be as a joint result of the increase in interconnect traffic and increase in program run time.

6.5 Further Work

With such a large number of variables it is possible to consider further combinations that were outside the scope of this chapter. Other than simply considering large or smaller sizes or pairings, it would be more interesting to try combining further different types of branch predictors. This could be done by simply looking at increasing the number of sizes, up to a different size per BPU, or by considering totally different predictors altogether, such as TAGE or Perceptrons.

Further consideration could be given to looking at different types of heterogeneity within the cores, whether by changing the instruction cache sizes or issue width. These have previously been shown to have an impact on how large an impact the BPU can have on the performance of the processor as a whole [63]. As a result, altering these parameters may make different BPU strategies either more or less effective.

Despite the encouraging prediction accuracy results shown in 6.2, the cycle and energy results in figures 6.3 and 6.4 suggest that sharing updates is impacting too heavily on the performance of the interconnect. The scheme used here is the same as was settled upon in chapter 5. However, this chapter only conducted research into different communication strategies for the 64K entry predictors. The communications strategy of what is worth communicating is the most important aspect of sending updates between BPUs, therefore it would be worthwhile returning to consider if the best communications strategy is different for the type of smaller predictors assessed in this chapter.

6.6 Summary

This chapter has explored the idea of choosing an architecture with a heterogeneous approach to BPU selection. This builds upon the solution presented in chapter 5 where each of the BPUs communicates with each other BPU to increase the prediction accuracy obtainable for data-parallel workloads.

The results have shown that, by choosing the right kind of heterogeneity, the size of the cache required for the BPU can be reduced by up to 35%, while increasing mispredictions by less than 1%. By making use of communication between BPUs to further reduce branch mispredictions, with up to a 4% reduction in miss rate. The results also showed that sharing updates between cores can allow for miss rates to be improved by up to 2.5% while reducing the bits budget by up to 25%.

This is an important result for embedded devices, or other processors bound by energy or die-space requirements, as it will serve to increase battery life while also improving performance. Alternatively the same solution could to used to increase the number of cores that can fit into the same die-space due to the reduction in die-space required for the BPU caches.

Chapter 7

Conclusion

This thesis has been centred around exploring two questions presented in section 1.9.1, can BPUs for embedded processors can be improved and what is the best way to modify existing BPUs technologies to target the needs of a given processor.

This chapter summarises the major contributions of this thesis from chapters 4 to 6. The contributions are then brought together and analysed to answer the questions posed in section 1.9.1 and what the limitations of the contributions are. Finally, consideration is given to further work arising from the results presented in this thesis.

7.1 Contributions

7.1.1 Hybrid Dynamic-Static Predictors For Embedded Processors

Chapter 4 demonstrated that the introduction of a hybrid predictor can be made possible for even the most restrictive of die space and energy requirements found on modern embedded processors. Furthermore, through careful consideration of how such a hybrid predictor is constructed the result is an overall increase in performance at a very low increase in power consumption.

A new bias parameter was introduced, allowing for a small fraction of an application's performance to be traded off for energy efficiency. The results showed a performance change of 7 - 8% with a change in peak performance of only 0.5% when moving from a bits budget of 4.125KB to 0.258KB. Furthermore, up to 90% of the branches are predicted statically, giving a very low energy overhead for the dynamic predictor.

7.1.2 Cooperative Branch Prediction For Multicore Data-parallel Workloads

Chapter 5 presented a technique that aimed to exploit the similarities in branch outcomes across multiple cores executing the same data-parallel application on different data points. The results in figure 5.17 showed an average reduction in misprediction rate of 1%, up to a maximum of 25%, accompanied by a reduction in run time of an average of 1% and up to 6%.

The energy model based on McPAT suggested that this is achieved at no extra energy cost. The approach of using the data bus by simply connecting it to the BPUs resulted in a minimum of additional hardware complexity. Finally, the technique was demonstrated to be suitable for a range of different branch predictor sizes, an explanation was provided as to how and why it is successful. In doing so it was shown that a small BPU using the technique can outperform a much larger BPU that does not use the technique.

7.1.3 Heterogeneous Branch Predictors For Reduced Energy/Die-space

Chapter 6 extended the work from chapter 5, by considering an architecture with a heterogeneous approach to BPU selection. The results in figure 6.2 showed that by choosing the right kind of heterogeneity the size of the cache required for the BPU can be reduced and the accuracies achieved increased. This was shown to be true for processors that simply make use of differently sized BPUs, but was more useful for processors making use of communication between BPUs to further reduce branch mispredictions.

7.2 Summary

In answering the question of can BPUs for embedded processors can be improved, chapter 4 showed that it is possible for embedded processors to be able to make use of a small dynamic predictor along-side static predictors, incurring a small extra die-space but improving performance by up to 8%. Alternatively, the same approach could be used to preserve performance and reduce predictor bits budget by a factor of 16X. As discussed in chapter 1 (sections 1.6 and 1.7), the constraints of energy, power and die-

space are of chief concern to embedded processors and will only grow in importance in the future.

The technique presented in chapter 4 holds promise in helping architects in making the design decisions necessary to balance these increasingly important requirements. Figure 4.4 shows how even a very small dynamic predictor can outperform a state of the art static predictor, while figure 4.5 shows that for this to happen it is important to only predict branches dynamically if they really require the extra accuracy of a dynamic BPU. A second important result in figure 4.4 is that a much smaller BPU can perform similarly to much larger BPUs in the right circumstances, making dynamic predictors viable in the highly constrained die-space requirements of embedded processors.

In taking a largely unmodified BPU and sharing information, chapters 5 and 6 answered the second question of this thesis and found a novel way of using the existing highly accurate BPU technologies to target the needs of a given processor. In figures 5.20 and 6.2 the results show that data-parallel applications can be improved by sharing information between the BPUs, but only for certain benchmarks and at certain bit budgets.

Chapter 6 showed that the right heterogeneous configuration can have a large impact on the bits budget required for the BPU while having little negative impact on the performance. Figure 6.3 showed that sharing updates can be sucessful for heterogeneous processors, while figure 6.4 demonstrated the powerful effects of heterogeneity on the energy consumed.

7.3 Analysis

One disadvantage of the techniques introduced in chapters 4, 5 and 6 is that they are based on static profiling carried out ahead of time. This means that they must be profiled for each combination of application and hardware configuration that the techniques are to be applied to decide when, if and how the technique should be used.

Additionally, the techniques are unable to react to an input set that is dramatically different to the training set used. As a result the techniques may end up reducing performance where they had been expected to improve performance. A way to dynamically control when and how the techniques are applied would allow for this to be addressed. Such a technique was outside the scope of this thesis but possible ways of achieving this are discussed in 7.4.

Chapters 5 and 6 made some attempt at detailing how it would be possible to im-

plement the techniques in a real processor. However, the precise details of how this can be achieved can only be shown through a lower level exploration, such as a register or gate level model. Such a model would also be able to give more accurate energy figures for the updates being sent across the interconnect. The results in section 6.4 have shown that the energy used by these updates is an important factor in deciding whether or not the technique will reduce energy consumption. Presenting such a model is beyond the scope of this thesis as it was chosen that the focus should be on more of a limit study of how and when the technique may be useful, rather than getting bogged down in the specifics of one particular implementation.

7.4 Future Work

There are several interesting directions that it would be worth extending the work in chapters 5 and 6. The results in these chapters have shown that while sharing information between BPUs can be helpful at some times, but harmful at others. To ensure that the correct information is shared at the right time a method of dynamically enabling and disabling sharing should be developed.

Remote Update Predictor Table

This could take several different forms, perhaps the most straight-forward would be to modify the hybrid GShare predictor presented in chapters 5 and 6 by adding a new predictor table. This table would only be updated by information sent from other BPUs and would be accessed in parallel to the existent GShare and Bimodal predictors with a new or extended meta table to choose between using local or remote updates.

Centralised/Distributed Update Controller

Another method would be to introduce some structure responsible for monitoring when updates are useful and what type of updates are most useful. If this were a centralised structure it could also be possible to reduce traffic by collecting together similar updates to ensure that redundant messages are not transmitted. This could also be distributed into some form of specialised interconnect. This may help eliminate messages earlier, but the reduced information could lead to eliminating updates that may be useful and could use more die-space.

The Controller could also be used pro-actively to send signals out to BPUs that are producing unhelpful updates to stop them them from sending any. Some mechanism could then be used to signal when it may be useful for the BPU to restart sending updates.

Alternative Interconnect Topology

Chapters 5 and 6 considered sending updates across a split address/data bus used for the existing instruction and data caches. However, it would be interesting to investigate the effect of a different topology on the effectiveness of shared updates. For instance, a bi-directional ring could be used for faster updates to neighbouring BPUs, or a tree structure could be used to filter updates, performing the function of a Distributed Update Controller.

Bibliography

[1] Kaveh Aasaraai and Amirali Baniasadi. A power-aware alternative for the perceptron branch predictor. In *Proceedings of the 12th Asia-Pacific Conference on Advances in Computer Systems Architecture*, ACSAC'07, pages 198–208, Berlin, Heidelberg, 2007. Springer-Verlag.

[2] ARM Ltd. ARM Cortex M3, 2011.

[3] Thomas Ball and James R. Larus. Branch prediction for free. *SIGPLAN Not.*, 28:300–313, June 1993.

[4] A. Baniasadi and A. Moshovos. Branch predictor prediction: a power-aware branch predictor for high-performance processors. In *Computer Design: VLSI in Computers and Processors, 2002. Proceedings. 2002 IEEE International Conference on*, pages 458–461, 2002.

[5] Amirali Baniasadi and Andréas Moshovos. Sepas: A highly accurate energy-efficient branch predictor. In *Proceedings of the 2004 International Symposium on Low Power Electronics and Design*, ISLPED '04, pages 38–43, New York, NY, USA, 2004. ACM.

[6] Christian Bienia. *Benchmarking Modern Multiprocessors*. PhD thesis, Princeton University, January 2011.

[7] Maximilien Breughe, Zheng Li, Yang Chen, Stijn Eyerman, Olivier Temam, Chengyong Wu, and Lieven Eeckhout. How sensitive is processor customization to the workloads input datasets?, 2011.

[8] Martijn Briejer, Cor Meenderinck, and Ben Juurlink. Extending the Cell SPE with energy efficient branch prediction. In *Proceedings of the 16th International Euro-Par Conference on Parallel Processing: Part I*, EuroPar'10, pages 304–315, Berlin, Heidelberg, 2010. Springer-Verlag.

[9] Ioana Burcea and Andréas Moshovos. Phantom-BTB: a virtualized branch target buffer design. In *Proceeding of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '09, pages 313–324, 2009.

[10] Brad Calder, Dirk Grunwald, Michael Jones, Donald Lindsay, James Martin, Michael Mozer, and Benjamin Zorn. Evidence-based static branch prediction using machine learning. *ACM Trans. Program. Lang. Syst.*, 19:188–222, January 1997.

[11] Po-Yung Chang et al. Branch classification: a new mechanism for improving branch predictor performance. In *Proceedings of the 27th Annual International Symposium on Microarchitecture*, MICRO 27, pages 22–31, 1994.

[12] Yen-Jen Chang. Lazy BTB: reduce BTB energy consumption using dynamic profiling. In Fumiyasu Hirose, editor, *ASP-DAC*, pages 917–922. IEEE, 2006.

[13] Daniel Chaver et al. Branch prediction on demand: an energy-efficient solution. In *Proceedings of the 2003 international symposium on Low power electronics and design*, ISLPED '03, pages 390–395, 2003.

[14] Bumyong Choi, Leo Porter, and Dean M. Tullsen. Accurate branch prediction for short threads. In *ASPLOS'08*, 2008.

[15] Michele Co, Dee A. B. Weikle, and Kevin Skadron. A break-even formulation for evaluating branch predictor energy efficiency. In *Proceedings of the Workshop on Complexity-Effective Design*, pages 38–49, 2005.

[16] Michele Co, Dee A.B. Weikle, and Kevin Skadron. A break-even formulation for evaluating branch predictor energy efficiency. In *32nd Annual ACM/IEEE International Symposium on Computer Architecture*, 2005.

[17] Standard Performance Evaluation Corporation. The standard performance evaluation corporation, 2014.

[18] Brian L. Deitrich, Ben chung Cheng, and Wen mei W. Hwu. Improving static branch prediction in a compiler. In *Proceedings of the 1998 International Conference on Parallel Architectures and Compilation Techniques*, PACT '98, 1998.

[19] A. Eden, J. Ringenberg, S. Sparrow, and T. Mudge. Hybrid myths in branch prediction. In *Conf. on Information Systems Analysis and Synthesis (ISAS 2001)*, volume volume XIV of *Comp.Sci. & Engineering*, pages 74–81, July 2001.

[20] A. N. Eden and T. Mudge. The YAGS branch prediction scheme. In *Proceedings of the 31st annual ACM/IEEE international symposium on Microarchitecture*, MICRO 31, pages 69–77, Los Alamitos, CA, USA, 1998. IEEE Computer Society Press.

[21] Marius Evers, Po-Yung Chang, and Yale N. Patt. Using hybrid branch predictors to improve branch prediction accuracy in the presence of context switches. In *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, ISCA '96, pages 3–11, New York, NY, USA, 1996. ACM.

[22] DongRui Fan, HongBo Yang, GuangRong Gao, and RongCai Zhao. Evaluation and choice of various branch predictors for low-power embedded processor. *J. Comput. Sci. Technol.*, 18(6):833–838, November 2003.

[23] S. Farling. Combining branch predictors. Technical Report TN-36m, Digital Western Research Laboratory, June 1993.

[24] Hongliang Gao and Huiyang Zhou. An effective way to improve perceptron predictors. In *The 1st JILP Championship Branch Prediction Competition (CBP-1)*, 2004.

[25] Hongliang Gao and Huiyang Zhou. Adaptive information processing: An effective way to improve perceptron branch predictors. *Journal of Instruction-Level Parallelism*, 7, 2005.

[26] Gartner. Gartner says worldwide PC, tablet and mobile phone shipments to grow 5.9 percent in 2013 as anytime-anywhere-computing drives buyer behavior, 2013.

[27] Wei hau Chiao and Chung ping Chung. Filtering of unnecessary branch predictor lookups for low-power processor architecture. *Journal of Information Science and Engineering*, 24:1127–1142, 2008.

[28] John L. Hennessy and David A. Patterson. *Computer Architecture A Quantatitative Approach*. Morgan Kaufmann, 2007.

[29] Michael Hicks, Colin Egan, Bruce Christianson, and Patrick Quick. Towards an energy efficient branch prediction scheme using profiling, adaptive bias measurement and delay region scheduling. In *International Conference on Design & Technology of Integrated Systems in Nanoscale Era*, 2007.

[30] Zhigang Hu, P. Juang, K. Skadron, D. Clark, and M. Martonosi. Applying decay strategies to branch predictors for leakage energy savings. In *Proceedings of the 2002 IEEE International Conference on Computer Design:* VLSI *in Computers and Processors (ICCD'02)*, ICCD '02, pages 442–, Washington, DC, USA, 2002. IEEE Computer Society.

[31] Daniel A. Jiménez, Stephen W. Keckler, and Calvin Lin. The impact of delay on the design of branch predictors. In *Proceedings of the 33rd annual ACM/IEEE international symposium on Microarchitecture*, MICRO 33, pages 67–76, New York, NY, USA, 2000. ACM.

[32] Daniel A. Jiménez and Calvin Lin. Dynamic branch prediction with perceptrons. In *Proceedings of the 7th International Symposium on High-Performance Computer Architecture*, HPCA '01, pages 197–, Washington, DC, USA, 2001. IEEE Computer Society.

[33] Daniel A. Jimnez. 2nd JILP championship branch prediction CBP-2, 2006.

[34] Yu-ping Jin, Hai-jian Sun, and Ping Jiao. The research of multi-core architectures's predictor. In *Computational Intelligence and Natural Computing Proceedings (CINC), 2010 Second International Conference on*, pages 145–148, 2010.

[35] Jkup Justinussen. 8 smartphones for every 5 pcs.

[36] Roger Kahn and Shlomo Weiss. Thrifty BTB: A comprehensive solution for dynamic power reduction in branch target buffers. *Microprocess. Microsyst.*, 32(8):425–436, November 2008.

[37] Sangwook P. Kim and Gary S. Tyson. Analyzing the working set characteristics of branch execution. In *31st International Symposium on Microarchitecture*, 1998.

[38] Soontae Kim, N. Vijaykrishnan, M. J. Irwin, and L. K. John. On load latency in low-power caches. In *ISLPED '03 Proceedings of the 2003 international symposium on Low power electronics and design*, 2003.

[39] Nadav Levison and Shlomo Weiss. Branch target buffer design for embedded processors. *Microprocess. Microsyst.*, 34(6):215–227, October 2010.

[40] Nadav Levison and Shlomo Weiss. Low power branch prediction for embedded application processors. In *Proceedings of the 16th ACM/IEEE International Symposium on Low Power Electronics and Design*, ISLPED '10, pages 67–72, New York, NY, USA, 2010. ACM.

[41] Contributor Forbes Louis Columbus. Idc: 87% of connected devices sales by 2017 will be tablets and smartphones, 2013.

[42] Srilatha Manne, Artur Klauser, and Dirk Grunwald. Pipeline gating: Speculation control for energy reduction. *SIGARCH Comput. Archit. News*, 26(3):132–141, April 1998.

[43] Marss tutorial. In *ISCA*, 2012.

[44] Rodney Van Meter. Caching and memory hierarchy, virtual memory, 2012.

[45] Pierre Michaud, André Seznec, and Richard Uhlig. Trading conflict and capacity aliasing in conditional branch predictors. *SIGARCH Comput. Archit. News*, 25(2):292–303, May 1997.

[46] M. Monchiero et al. Power-aware branch prediction techniques: a compiler-hints based approach for VLIW processors. In *Proceedings of the 14th ACM Great Lakes symposium on VLSI*, GLSVLSI '04, pages 440–443, 2004.

[47] Karthik Natarajan, Heather Hanson, Stephen W. Keckler, Charles R. Moore, and Doug Burger. Microprocessor pipeline energy analysis. In *Proceedings of the 2003 International Symposium on Low Power Electronics and Design*, ISLPED '03, pages 282–287, New York, NY, USA, 2003. ACM.

[48] G. Palermo, M. Sam, C. Silvan, V. Zaccari, and R. Zafalo. Branch prediction techniques for low-power VLIW processors. In *Proceedings of the 13th ACM Great Lakes Symposium on VLSI*, GLSVLSI '03, pages 225–228, New York, NY, USA, 2003. ACM.

[49] Dharmesh Parikh, Kevin Skadron, Yan Zhang, Marco Barcella, and Mircea R. Stan. Power issues related to branch prediction. In *Proceedings of the 8th International Symposium on High-Performance Computer Architecture*, HPCA '02, pages 233–, Washington, DC, USA, 2002. IEEE Computer Society.

[50] Dharmesh Parikh, Kevin Skadron, Yan Zhang, and Mircea Stan. Power-aware branch prediction: Characterization and design. *IEEE Trans. Comput.*, 53(2):168–186, February 2004.

[51] Sudeep Pasricha and Alex Veidenbaum. Improving branch prediction accuracy in embedded processors in the presence of context switches. In *21st International Conference on Computer Design*, 2003.

[52] Avadh Patel, Furat Afram, Shunfei Chen, and Kanad Ghose. MARSSx86: A Full System Simulator for x86 CPUs. In *Design Automation Conference 2011 (DAC'11)*, 2011.

[53] Harish Patil and Joel S. Emer. Combining static and dynamic branch prediction to reduce destructive aliasing. In *High-Performance Computer Architecture, 2000. HPCA-6. Proceedings. Sixth International Symposium on*, pages 251 –262, 2000.

[54] Peter Petrov and Alex Orailoglu. Low-power branch target buffer for application-specific embedded processors. In *Proceedings of the Euromicro Symposium on Digital Systems Design*, DSD '03, pages 158–, Washington, DC, USA, 2003. IEEE Computer Society.

[55] Michael D. Powell, Amit Agarwal, T. N. Vijaykumar, Babak Falsafi, Babak Falsafi , and Kaushik Roy. Reducing set-associative cache energy via way-prediction and selective direct-mapping, 2001.

[56] Alex Ramirez, Josep L. Larriba-Pey, and Mateo Valero. Branch prediction using profile data, 2001.

[57] Behnam Robatmili, Dong Li, Hadi Esmaeilzadeh, Sibi Govindan, Aaron Smith, Andrćw Putnam, Doug Burger, and Stephen W. Keckler. How to implement effective prediction and forwarding for fusable dynamic multicore architectures. In *Proceedings of the 2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA)*, HPCA '13, pages 460–471, Washington, DC, USA, 2013. IEEE Computer Society.

[58] Es So Rs, Zach Purser, Karthik Sundaramoorthy, and Eric Rotenberg. A study of slipstream processors. In *Proceedings of the 33rd Annual International Symposium on Microarchitecture*, pages 269–280. ACM Press, 2000.

[59] R. Sendag, J.J. Yi, Peng fei Chuang, and D.J. Lilja. Low power/area branch prediction using complementary branch predictors. In *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*, pages 1 –12, april 2008.

[60] André Seznec. Revisiting the perceptron predictor. *IRISA*, 1620, 2004.

[61] André Seznec. The L-TAGE branch predictor. *J. Instruction-Level Parallelism*, 9, 2007.

[62] André Seznec and Pierre Michaud. A case for (partially) TAgged GEometric history length branch prediction. *J. Instruction-Level Parallelism*, 8, 2006.

[63] Kevin Skadron, Pritpal S. Ahuja, Margaret Martonosi, and Douglas W. Clark. Branch prediction, instruction-window size, and cache size: Performance tradeoffs and simulation techniques. *IEEE Trans. Comput.*, 48(11):1260–1281, November 1999.

[64] James E. Smith. A study of branch prediction strategies. In *Proceedings of the 8th Annual Symposium on Computer Architecture*, ISCA '81, pages 135–148, 1981.

[65] Karthik Sundaramoorthy, Zach Purser, and Eric Rotenburg. Slipstream processors: Improving both performance and fault tolerance. *SIGARCH Comput. Archit. News*, 28(5):257–268, November 2000.

[66] Synopsys, Inc. DesignWare ARC EM6 Processor Core, 2013.

[67] Mingxing Tan, Xianhua Liu, Zichao Xie, Dong Tong, and Xu Cheng. Energy-efficient branch prediction with compiler-guided history stack. In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '12, pages 449–454, San Jose, CA, USA, 2012. EDA Consortium.

[68] Weiyu Tang, Rajesh Gupta, Alexandru Nicolau, and Alexander V. Veidenbaum. Simultaneous way-footprint prediction and branch prediction for energy savings in set-associative instruction caches. In *IEEE workshop on power management for real-time and embedded systems*, 2001.

[69] David Tarjan and Kevin Skadron. Merging path and gshare indexing in perceptron branch prediction. *ACM Trans. Archit. Code Optim.*, 2:280–300, September 2005.

[70] EEMBC benchmark suite, 2004. http://www.eembc.org/home.php.

[71] Shyamkumar Thoziyoor et al. HP Labs: CACTI, 2010.

[72] Nigel Topham and Daniel Jones. High speed cpu simulation using JIT binary translation. In *The 3rd Annual Workshop on Modeling, Benchmarking and Simulation*, 2007.

[73] S. Verma, B. Maderazo, and D.M. Koppelman. Spotlight - a low complexity highly accurate profile-based branch predictor. In *2009 IEEE 28th International Performance Computing and Communications Conference (IPCCC*, pages 239–247. IEEE, 2009.

[74] Tangm Weiyu, Alexander V. Veidenbaum, Alexandru Nicolau, and Rajesh Gupta. Integrated i-cache way predictor and branch target buffer to reduce energy consumption. In *Proc. of the Int. Symp. on High Performance Computing, Springer LNCS 2327*, pages 120–132, 2002.

[75] I-Wei Wu, Bin-Hua Tein, and Chung-Ping Chung. Instruction fetch power reduction using forward-branch and subroutine bufferable innermost loop buffer. In *2006 ICS International Computer Conference*, Dept. of Computer Science, National Chiao Tung University, 2007.

[76] Chengmo Yang and Alex Orailoglu. Power efficient branch prediction through early identification of branch addresses. In *Proceedings of the 2006 international conference on Compilers, architecture and synthesis for embedded systems*, CASES '06, pages 169–178, New York, NY, USA, 2006. ACM.

[77] Tse-Yu Yeh and Yale N. Patt. Alternative implementations of two-level adaptive branch prediction. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, ISCA '92, 1992.

[78] Tse-Yu Yeh and Yale N. Patt. A comparison of dynamic branch predictors that use two levels of branch history. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 257–266. ACM, 1993.

[79] Cliff Young and Michael D. Smith. Static correlated branch prediction. *ACM Trans. Program. Lang. Syst.*, 21(5):1028–1075, September 1999.

[80] Po yung Chang, Eric Hao, and Yale N. Patt. Alternative implementations of hybrid branch predictors. In *Proceedings of the 28th Annual International Symposium on Microarchitecture*, pages 252–257, 1995.

[81] Ruijian Zhang, Willis K. King, and Manhong Guo. A hybrid branch prediction scheme - an integration of software and hardware techniques. In *The Mid-Atlantic Student Workshop on Programming languages and Systems*, 2001.

[82] Wei Zhang and Bramha Allu. Reducing branch predictor leakage energy by exploiting loops. *ACM Trans. Embed. Comput. Syst.*, 6(2), May 2007.

[83] Ming-Yuan Zhong and Jong-Jiann Shieh. Power improvement using block-based loop buffer with innermost loop control. In *Proceedings of the 10th International Conference on Algorithms and Architectures for Parallel Processing - Volume Part II*, ICA3PP'10, pages 368–380, Berlin, Heidelberg, 2010. Springer-Verlag.